C78 - Computer Algebra

Russell Bradford

1 Introduction

Question 1: what is the value of

$$\int \frac{1}{x^8 - 1} dx?$$

Question 2: what is the coefficient of x^{14} in the Taylor expansion of $\sin\cos\tan x$ about 0?

Answer 1: it is

$$\begin{aligned}\big(&\sqrt{2}\log(-\sqrt{2}x + x^2 + 1) - \sqrt{2}\log(\sqrt{2}x + x^2 + 1) \\ &+2\sqrt{2}\arctan(1 - \sqrt{2}x) - 2\sqrt{2}\arctan(1 + \sqrt{2}x) \\ &+2\log(x-1) - 2\log(x+1) - 4\arctan x\big)/16\end{aligned}$$

Answer 2: it is $(348874344\sin(1) - 1765148591\cos(1))/87178291200$.

It's possible to do these by hand, but it's much nicer to get a computer to do it for you. A machine did the above integral in much less than a second.

Of course we could have made these examples *much* bigger. For example, try the integral $\int x^n e^x \, dx$ which has $n+1$ terms, the largest of which has coefficient $n!$. This illustrates a general principle in CA: we don't know how large the answer is going to be until we see it! This is in strong contrast to the more usual number crunching approach to problems—if we do an integral numerically we know exactly how large (in terms of data size) the problem is, but if we do an integral *symbolically* then there is no way of predicting in advance how much store we will need during the computation, or how much will be needed to store the result.

1.1 What is Computer Algebra?

A partial definition (as are all definitions of subjects):

Computer Algebra is constructive algebra plus efficient algorithms.

This means not only do we have to find constructive proofs of theorems in algebra, we also need to implement those constructions in such a way that we don't have to wait a few eons for the answer. For example, we can factorize every integer by the trial division method—but just try factorizing a random 100 (or 200, or 300 ...) digit number that way. We use more sophisticated techniques (these may have been covered in a number theory course).

This course ought to be titled "Computational Algebra," i.e., algebra that is done on computer, but the above name has stuck. This name implies there could be courses with name like "Computer Analysis"—no-one seems to have progressed very far with this one, or like "Computer Logic"—those of you who attended C80 already know a bit about this. The absence of Computer Analysis is not so bad as one might think—many analytic problems can be viewed as algebraic ones, e.g., differentiation and integration of polynomials can be thought of in purely algebraic terms. In fact, there is a branch of algebra named "Differential Algebra" which concerns itself with just this sort of problem. We shall be tackling a bit of this later in the course.

Now even "Computational Algebra" is too generous a term for what we are going to cover: if we are lucky we might touch upon something less basic than arithmetic and school-level calculus. It is a constant surprise (to me, at any rate) of the range of subtle mathematics required to find efficient algorithms to do those problems we idled our time away with in the sixth-form.

The concept of algebraic, in contrast to numeric, computation has been around for a long time. Indeed, this is how many of the Victorian mathematicians spent their entire working lives. The most famous story is that of Delaunay, who worked on a formula for the orbit of the moon: he took twenty years to produce a 128-page formula, ten years to calculate, and ten years to check. Legend has it that when the first computer algebra systems were set on the problem, they took just a few minutes to produce the same result. (Apparently Delaunay made only two mistakes, the second of which was a consequence of the first).

Nowadays, what with communications (and other) satellites, this sort of computation is made every day. It appears that industry is slowly becoming aware of the possibilities of symbolic computation, we get inquiries from all sorts of bodies. And business is not far behind—now there are "commercial" algebra systems, of which at least one is specifically designed to make money by having a high gloss interface hiding some quite dodgy coding.

In the purer world of the ivory tower, general relativists use algebra systems heavily to compute with space-time metrics: a large part of the classification of metrics is only possible with the power of a computer. In my own field, number theory, CA is starting to emerge as a useful tool as it is capable of manipulating polynomials, large integers, and general algebraic numbers to infinite precision. Thus $\sqrt{2}$ is an object which, when squared, returns 2. There is none of this messy floating point rounding error nonsense.

There have been attempts in the past to circumvent the rounding error problem. For example, the exists an "infinite-precision" Fortran that performs all its computations to the required number of decimal places, no matter how many. Of course, it is much slower than real Fortran, but it excited many people as it allowed them to, say, invert ill conditioned matrices with some hope of getting the right answer. This is just one step along the path to symbolic computation.

There is an interplay, naturally, between numeric and symbolic computation. Often it is better to take a problem, recast or reform it using symbolic methods, and then solve the new problem numerically. Typically this would optimise expressions, or change poorly conditioned problems into better conditioned ones. There is ongoing research at Bath into the interface between symbolic algebra and numerical libraries such as NAG.

1.2 Capabilities

What can current algebra systems do? Typically some subset of

- Algebra: factor, expand and simplify expressions. Solve systems of algebraic equations. Manipulate trigonometric and other elementary functions. Manipulate matrices and tensors.
- Calculus: differentiate. Integrate (definite and indefinite). Take limits. Solve ordinary DEs. Manipulate and compute Taylor and Laurent series. Compute Fourier and Laplace transforms.
- Numerical analysis: compute with arbitrary precision arithmetic. Generate FORTRAN code.
- Programming environment: define functions and programs for complex algorithms.

- User environment: often neglected is "how easy is the system to use?" and things like i/o and window-based interactions.

There are also systems that doesn't conform to the above descriptions, namely, for example, the group-theory system Cayley. This works with groups and group elements as basic objects, and can answer questions like "given a group presentation, compute its centralizer, and find its order." And there are many other systems highly tuned to specific jobs. In the past it was worth a Ph.D. to write an algebra system. These days it is something you knock off in an afternoon.

1.3 Some Algebra Systems

Computer programs to do algebra broadly fall into two major classes: the first, the "algebra systems" attempts to be all things to all people by providing functions to to everything you could possibly think of. The second, the "specialized systems" are are geared to solving specific problems (e.g., classifications of space-time metrics in General Relativity) and are typically much smaller and much more efficient (in their field).

ALPACK		Precursor to ALTRAN
ALTRAN	Fortran?	Very old
FORMULA	ALGOL	
FORMAC		
Scratchpad		1966.
MATHLAB		1968. Numerical processing environment
CAMAL		Special relativity
REDUCE	Lisp	"Free," old, undergoing rejuvenation
Macsyma	Lisp	Semi-commercial, 1970, large
Scratchpad II	Lisp	1975. Very large amount of code, mathematically sophisticated
MuMath	Lisp	1979. Precursor of Derive
VAXIMA	Lisp	Really Macsyma
SMP	Fortran	1982. Very old, dead (zombie)
DOE-Macsyma	Lisp	
Maple	C	1985. Educational use, growing rapidly
MathCAD		1985. As Mathlab
Powermath	Pascal	1986. On a Mac
Derive	Lisp	1988. Educational, small. Good on a PC. Remarkable attention to detail
Mathematica	"C"	1988. Highly commercial, very new and growing rapidly (dodgy practices)
Theorist		1988. Pretty front end
PARI		Number Theory
FORM		1989. Number Theory
ALJABR		1991. Really Macsyma
Mathcad		1991.
Symbmath		1991.
Axiom	Lisp	Scratchpad reborn!
Paramacs	Lisp	Really Macsyma

And a host of other specialized systems: Sheep, Alkahest III, Schoonschip in special relativity; Reve, Cayley, Gap in group theory, MAO for celestial mechanics ...See the chapter by van Hulzen and Calmet in the Computing Supplementum 4 for all the gory details.

There is a local bias towards REDUCE, since many of us are or have been involved in writing parts of REDUCE, and towards Axiom for the same reasons. There is also Maple, which is particularly good for teaching purposes. We shall be using REDUCE and Maple, due to the large amount of local expertise and their availability on the hardware we have.

1.4 Outline of Course

This is subject to change without prior notification. The management takes no responsibility etc. In some particular order (but no particular emphasis) we have:

- Introduction to REDUCE
  - what it looks like and what it does
- Numbers
  - how to represent numbers
  - their arithmetic and GCD
- Polynomials
  - how to represent polynomials
  - their arithmetic and GCDs
  - Expression swell and modular methods
- Matrices
  - representation and arithmetic
- User algorithms: three examples of more complicated algorithms
  - Polynomial factorization
  - Algebraic Approximation
  - Taylor series
- Polynomial Simplification
  - Buchberger algorithm
  - Wu method
- Definite Integration
  - Liouville's theorem
  - Risch
  - Risch-Norman
- Miscellaneous stuff
  - Richardson's theorem
  - root finding
  - ...

1.5 Course books

The major text for this course will be "Computer Algebra," by J.H. Davenport, Y. Siret, and E. Tournier (Academic Press, 1988). This is a good introduction and overview, and has the major advantage of being cheap. (There is also a French language edition "Calcul Formel" if you feel that way inclined.)

A more advanced text is "Computer Algebra," edited by B. Buchberger, G.E. Collins, and R. Loos (Springer-Verlag Computing Supplementum 4, 1982), which goes into many of the details of the algorithms, although this is beginning to look a little dated. It is very good as an overview, though.

For the mathematics in this course, there is the excellent "Mathematics for Computer Algebra," by M. Mignotte, Springer-Verlag. Well worth a look.

No course in CA can get by without mentioning Knuth: "The Art of Computer Programming," vols. I–III, and in particular volume II, *Seminumerical Algorithms,* which is a source for many of the basic (and not-so-basic) algorithms.

Aho, Hopcroft and Ullman have the text "The Design and Analysis of Computer Algorithms," (Addison-Wesley), which covers some of the polynomial and matrix arithmetic algorithms.

On a more specific note, "REDUCE," by G. Rayna is a primer for the REDUCE algebra system, which we shall be using. A point to note is that this text is for REDUCE version 3.2, whereas we run REDUCE 3.4—the differences are minimal, but sometimes significant. This book is worth glancing at, but not buying.

A newer book is "Algebraic Computing with Reduce," by Malcolm MacCallum and Francis Wright, which is an in-depth introduction to the use of Reduce, but somewhat biased toward the solving of problems in differential equations.

Hot off the press is "Algorithms for Computer Algebra," by Geddes, Czapor and Labahn, published by Kluwer. For the rich or dedicated only: has a $140 price tag.

Another new book is "A Course in Computational Algebraic Number Theory," by Henri Cohen (Springer-Verlag GTM 138). This has the eponymous slant, but there is some stuff on polynomials and numbers.

There is also the REDUCE User's Manual—available from the school office—this deals exclusively with REDUCE, and details some of the more sophisticated things you can do in this system.

Proceedings for the yearly conferences in CA can be found in the Springer Lecture Notes in Computer Science series, namely the proceedings for conferences with names such as Eurocam, Eurosam, Eurocal, and ISSAC. Also the ACM have published the proceedings of their SYMSAC (Special Interest group on Symbolic and Algebraic Computation) conferences 1976, 1981, 1986, 1989 and 1990. These can be viewed as indicators of the current trends in research.

For those of you who already have this book, there is also some elementary bits and pieces in "Structure and Interpretation of Computer Programs," by H. Abelson and G. Sussman.

Many of these books may be found in the library (particularly the Springer LNCS series), but others are slightly more tricky to get hold of.

2 A Short Introduction to REDUCE

We have seen that REDUCE is just one of many algebra systems, but it is fairly representative of most of them in the style of programming. There are minor syntactic differences (e.g. REDUCE uses a semi-colon to delimit statements, where Axiom uses newlines), and functional differences (e.g. REDUCE expands and simplifies all expressions as a default, where Macsyma doesn't), but these are unimportant: most support the simple conversational format of question-answer, question-answer, and so on.

2.1 REDUCE at Bath

To run REDUCE simply type `reduce`.

You should get something like

```
REDUCE 3.4 (22 Oct 1991):

1:
```

The first command to remember is `bye;` This exits from the system. Every command is terminated by a semi-colon, and commands may be spread out over several lines. Spaces are only significant in the usual places.

You can type simple expressions

```
1: (x+1)^5;

 5      4       3       2
x  + 5*x  + 10*x  + 10*x  + 5*x + 1

2:
```

or

```
2: (sin x + cos y)^2/(tan x + 1);

       2                         2
 sin(x)  + 2*sin(x)*cos(y) + cos(y)
------------------------------------
             tan(x) + 1

3:
```

As a default, REDUCE does not know about the trigonometric relations such as $\sin^2 x + \cos^2 x = 1$, though we can add these later. There is an interesting theoretical difficulty at this point: when do we want to write $\sin^2 x$ as $1 - \cos^2 x$, and when vice-versa? REDUCE, initially, does not force the question. This is part of a much larger problem—what do we mean by simplification? Is $x^2 + 2x + 1$ simpler than $(x + 1)^2$? Or $x^{100} + 100x^{99} + \cdots + 1$ simpler than $(x + 1)^{100}$? Some systems, such as Macsyma and Maple provide many and varied expand and simplify operations, allowing the user to choose exactly which type of expansion to choose, and when to apply it.

Assignment: `f := sin x / x;` gives `f` the obvious value. Note that functions of one argument do not require parentheses, but functions of two or more arguments do: `df(f,x);` to differentiate `f` with respect to `x`. The expression `sin a*x` means the same as `sin(a)*x`. Variables that have not been assigned a value evaluate to themselves: thus `f + x` from above evaluates to Note:

```
             2
 sin(x) + x
-------------
      x


5: f := (x+y)^3;

      3      2           2    3
f := x  + 3*x *y + 3*x*y  + y

6: x := 2;

x := 2

7: f;

 3      2
y  + 6*y  + 12*y + 8

8:
```

To clear a definition for `x` (say), use `clear x;` and then `f;` again evaluates to

```
 3      2           2    3
x  + 3*x *y + 3*x*y  + y
```

Integration: `int(f,x);` integrates (the value of) `f` with respect to `x`. Lisp uses a load-on-need system, so that the first time an integration is performed there will be a slight pause while the package is loaded. From then on there should be less of a pause for the same problem. A constant of integration comes out in the wash of the integration algorithm, so that `int(df(f,x),x);` is not necessarily equal to `f;`.

Loops:
`for i := 1:100 do x := x + i;`
`for i := 1:100 sum i^2;`
`for i := 1:100 product i;` this is the factorial.
`for i := 1:10 product (x-i);` this computes $\prod_{i=1}^{10}(x-i)$.

Matrices: `m := mat((1,2),(x+y,23));` defines `m` to be a square matrix. This prints in different ways on the Orions and the Archimedes: the former echoes expressions of the type `m(1,1) := 1` and the latter

$$\begin{pmatrix} \mathtt{1} & \mathtt{2} \\ \mathtt{x}+\mathtt{y} & \mathtt{23} \end{pmatrix}$$

We can assign to individual elements by `m(2,2) := sin x;`. The determinant of a matrix is `det m;` its transpose `tp m;` arithmetic `m := m1 + m2*m3;` for matrices `m`i. The inverse of a matrix is `1/m;`, but take care over division: `m1/m2;` is different from `(1/m2)*m1`. Matrices must be conformant for the above to make sense. Multiplication by scalars also works. Thus

$$\mathtt{det}(\mathtt{m} - \mathtt{l} * \mathtt{mat}((\mathtt{1},\mathtt{0}),(\mathtt{0},\mathtt{1})));$$

computes the characteristic equation of `m`.

Each prompt bears a number, e.g. `5:` , and we use these to refer back to previous results. Thus `10 + ws(4)^2;` adds 10 to the square of result number 4. As a special case, the variable `ws` refers to the last result. ("ws" means "workspace"). To re-do calculation number 5, use `input 5;`.

To substitute values for variables in expressions without assigning to those variables, use `sub`. Thus

```
11: f;

 3      2           2    3
x  + 3*x *y + 3*x*y  + y

12: sub(x=3,f^2);

 6       5        4        3         2
y  + 18*y  + 135*y  + 540*y  + 1215*y  + 1458*y + 729

13: f;

 3      2           2    3
x  + 3*x *y + 3*x*y  + y

14:
```

This is most useful for computing limits and the like.

There is a more permenant (and powerful) type of substitution operator, called `let`. Writing `let r2^2 = 2;` means "whenever you see `r2` raised to a power of two (or greater) reduce it using this equation." Thus we now find things like `(x+r2)^3;` has a value `x^3 + 3*x^2*r2 + 6*x + 2*r2`. The variable `r2` is acting like $\sqrt{2}$. Rules can be more complicated, such as

```
for all x let (sin x)^2 = 1 - (cos x)^2;
```

This invokes the standard trigonometric rule whenever it sees a power of sin. The phrase "`for all x`" means that not only `sin x` (where `x` is regarded as a variable in its own right) is affected, but also powers of `sin y`, `sin(2*y + cos z)`, and so on, will all be reduced.

The `solve` operator tries to solve equations. To solve the simultaneous equations $x + y = a$ and $x - y = b$ for x and y, use

```
14: solve({x+y-a,x-y-b},{x,y});

     a + b     a - b
{{x=-------,y=-------}}
       2         2

15:
```

This is a list of lists of solutions: use `first ws;` to extract the first element of a list, similarly for the second and third. `part(f,`n`)` gives the n^{th} element of a list.

Solve currently can cope with linear simultaneous equations, and single higher degree equations: `solve(x^2+x+1,x);` finds the roots of the quadratic.

We can also define functions, a simple example being

```
15: procedure foo n;
if n < 2 then 1 else n*foo(n-1);

foo

16: foo 100;

933262154439441526816992388562667004907159682643816214685929638952175999932299
15608941463976156518286253697920827223758251185210916864000000000000000000000
0000

17:
```

And, of course, functions of several variables.

Modular arithmetic: the command `setmod` n`;` allows us to select a modulus to work by. Then `on modular;` informs REDUCE that the following computations are to be worked modulo n.

```
17: f := x^4+4;

      4
f := x  + 4

18: factorize f;

  2            2
{x  - 2*x + 2,x  + 2*x + 2}

19: setmod 5;

1

20: on modular;

21: f;

 4
x  + 4

22: factorize f;

{x + 1,x + 2,x + 3,x + 4}

23: foreach fact in ws product fact;

 4
X  + 4
```

```
24:
```

The function `setmod` returns the last value of the modulus. We use `off modular;` to re-instate computations over the integers.

The `foreach` construct allows us to loop down a list structure: `foreach ff in {1, 2, x, 4} sum ff;` returns `x + 7`.

To interrupt REDUCE, use `^C` as usual.

2.2 Packages

There are a many extra packages in REDUCE that are not pre-loaded, for example the power series package:

```
1: load_package 'tps;

TPS

2: ps(sin x, x, 0);

       1    3       1      5       7
{x - (---)*x  + (-----)*x  + O(x )}
       6           120
```

This computes the Taylor expansion of $\sin x$ in x about 0.

There is also the Gröbner basis package:

```
4: load_package 'groebner;

GROEBNER

5: groebner({x^2+y^2-a,x*y-b},{x,y});

      1   3     a
{x + ---*y  - ---*y,
      b         b

  4       2     2
 y  - a*y  + b }
```

For now, you can treat this as an algorithm that "triangularises" a set of simultaneous polynomial equations. Later in the course we shall go into this in detail.

2.3 More stuff

Read files by `in "file";` (use `end;` at EOF in the file). To output to a file `out "file";` after which all ouput goes to that file. Use `out t;` to redirect back to the termianl, and `shut "file"` to close it. The comment character is `%`.

There are various flags used to control expansion and output (amongst other things): `gcd`, `exp`, `nat`, `float`, `bigfloat`, `numval`, and so on. Use `on gcd;` and `off gcd;` as required.

To see the internal Lisp structure of an expression, use `Lisp ws;` but this may not be terribly enlightening!

2.4 Exercises (trivial)

Compute, using l'Hôpital's rule,

$$\lim_{x\to 0}\frac{x^3\sin x}{(1-\cos x)^2}.$$

Ans: 4

Compute $\sum_{n=1}^{50} n!$ *Ans:*

```
for n := 1:50 sum for i := 1:n product i;
```

31035053229546199656252032972759319953190362094566672920420940313

Prove by induction that

$$1^4+2^4+3^4+\ldots+n^4=n^5/5+n^4/2+n^3/3-n/30.$$

Evaluate $\int x^n e^x\,dx$ for $n=0\ldots10$.

Determine the first 10 coefficients of the Taylor expansion of $\log(x^3+3)\sin x$ about 0.

```
              log(3)     3      1    4      log(3)     5      1     6
log(3)*x - (--------)*x  + (---)*x  + (--------)*x  - (----)*x  -
                6              3           120             18
  log(3) + 280     7       1     8     log(3) + 3360    9       557      10
(--------------)*x  + (-----)*x  + (---------------)*x  + (-------)*x
      5040              360              362880              45360
     11
+ O(x   )
```

Confirm that $det(AB)=det(A)det(B)$ for the matrices

$$A=\begin{pmatrix}a_{11}&a_{12}&a_{13}\\a_{21}&a_{22}&a_{23}\\a_{31}&a_{32}&a_{33}\end{pmatrix}\quad\text{and}\quad B=\begin{pmatrix}b_{11}&b_{12}&b_{13}\\b_{21}&b_{22}&b_{23}\\b_{31}&b_{32}&b_{33}\end{pmatrix}$$

Do the same for their traces, and also the relevant rule for transposes.

Find the characteristic equation of the matrix

$$\begin{pmatrix}1&2&3&4\\x+1&y+2&z+3&w+4\\4&3&2&1\\x+4&y+3&z+2&w+1\end{pmatrix}$$

```
    3    2      2         2
l*(l  - l *w - l *y - 6*l  + 2*l*w - 6*l*x - 4*l*z - 40*l + 10*w

    - 10*x + 30*y - 30*z)
```

The $n \times n$ Hilbert matrix $H_n = (h_{ij}) = (1/(i+j-1))$ are numerically ill-conditioned. Compute their inverses for $n = 1 \ldots 10$.

What conditions on a and b ensure that the circle $x^2 + y^2 - a^2 = 0$ intersects the parabola $y^2 - x - 4b$?

Solve the differential equation

$$y' + (5/x + 4/x \log x)y = x^2 + (\log x)^3$$

```
                 7                 6                 5
(x*(497664*log(x)  - 580608*log(x)  + 580608*log(x)
                  4  2              4               3  2
     + 373248*log(x) *x  - 483840*log(x)  - 186624*log(x) *x
                  3                2  2                2
     + 322560*log(x)  + 69984*log(x) *x  - 161280*log(x)
                   2                         2
     - 17496*log(x)*x  + 53760*log(x) + 2187*x  - 8960))/(2985984
          4
   *log(x) )
```

(for the differentially challenged: $y' + fy = g$ has a solution $y = e^{-\int f} \int e^{\int f} g$.)

Generate some Legendre polynomials using a) the recurrence relation

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0,$$

with $P_0(x) = 1$ and $P_1(x) = x$, and b) Rodrigues' formula

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n.$$

Show, for a few examples, that

$$\int_{-1}^{1} P_i(x) P_j(x) = 2\delta_{ij}/(2i+1).$$

Compute some Bernoulli numbers B_n, defined by

$$\sum_{n=0}^{\infty} \frac{B_n}{n!} x^n = \frac{xe^x}{e^x - 1}.$$

Write a procedure `hopital(fun,var,lim)`, where `fun` is some function such that its denominator is zero when the variable `var` takes the value `lim`, that computes the required limit.

(Partial solution)

```
procedure hopital(fun,var,lim);
begin scalar n,d,valn,vald;
  n := num fun;
```

```
  d := den fun;
  valn := sub(var=lim,n);
  vald := sub(var=lim,d);
  while valn = 0 and vald = 0 do <<
    n := df(n,var);
    d := df(d,var);
    fun := n/d;
    n := num fun;
    d := den fun;
    valn := sub(var=lim,n);
    vald := sub(var=lim,d)
  >>;
  if vald = 0 then write "no value"
  else return valn/vald
end;
```

3 Integers

In this, and the next few, chapters we are going to consider the workings "under the bonnet" of algebra systems in general, and REDUCE in particular.

3.1 Requirements of an algebra system

If we are to construct an algebra system, we first face the question "what do we want the algebra system to do?" We have seen that we require manipulation of polynomials, rational functions, and transcendental functions. We might also require algebraic functions (e.g. $\sqrt{x^2-1}$) if we are being more adventurous. One thing that is absolutely necessary is the easy manipulation of very large integers, usually called "bignums" for whimsical reasons.

Let us take this as a starting point in an investigation into the internals of an algebra system: bignums, their implementation and arithmetic.

3.2 Bignums

In traditional computer languages there is the so-called integer type, but really they do not represent integers, but just some small range of integers, typically bounded by some power of two. However, in CA we often need large, completely accurate representations of numbers, of sometimes hundreds of thousands of digits long. Thus we have "bignums," which as the name suggests are arbitrarily large integers, where we have exact knowledge of *all* the digits.

A few examples proves that numbers can get very large very easily in even a seemingly trivial computation. Here is a simple polynomial gcd (ex DST)

$$\begin{aligned} A(x) &= x^8+x^6-3x^4-3x^3+8x^2+2x-5 \\ B(x) &= 3x^6+5x^4-4x^2-9x+21 \end{aligned}$$

Using the naïve Euclidean algorithm and clearing fractions we get

$$-15x^4 + 3x^2 - 9$$
$$15795x^2 + 30375x - 59535$$
$$1254542875143750x - 1654608338437500$$
$$12593338795500743100931141992187500$$

(2 digits, 5 digits, 16 digits, 35 digits) Thus they are coprime, but it has taken a lot of effort to determine this one bit answer. There are better algorithms for the above computation, but nevertheless we see the need to be able to manipulate arbitrarily large integers.

3.2.1 Representation

Bignums are almost always implemented in one particular way, namely as vectors of integers, which are the "digits" base b for some suitable b. It is also possible to use a list of integers, but then the individual digits are harder to access, and the representation is less compact (there do exist such implementations, and at least one is quite efficient). Thus we want the vector

$$[\mathtt{d_0}, \mathtt{d_1}, \mathtt{d_2}, \ldots, \mathtt{d_{n-1}}] \leftrightarrow d_0 + d_1 b + d_2 b^2 + \ldots + d_{n-1} b^{n-1}$$

What about the base b? There are two (sensible) choices, $b = 10^m$ or $b = 2^m$, where m is chosen so the powers fit nicely into a machine word. Typical values are 10^8 and 2^{31} for 32-bit words. The choice is one of print time against compute time. A base which is a power of 10 is easy to convert to and from a decimal representation, in particular we need to do this when printing out an answer. Arithmetic on the digits is fairly easily, apart from computing the carry on to the next digit, as this requires divisions by 10. On the other hand a power of 2 maps neatly on to the binary arithmetic of the underlying machine, and carries are handled by the hardware. However, converting a several-thousand-binary-digit number to decimal can take quite a while, involving a large number of divisions by 10.

Moral: if you are doing lots of computation with large numbers, but small or infrequent i/o, use a power of 2 for your base. If you like looking at bignums, use a power of 10.

3.2.2 Arithmetic

Addition and subtraction are trivial: use the obvious algorithms, carrying over at each step. Even Knuth agrees there is little we can do to improve these algorithms (on a uniprocessor machine...).

For multiplication we can use the school tabular method: write down the cross-products of the digits and sum. If we are working with n digit numbers, this requires $O(n^2)$ elementary operations (i.e. single-digit multiplications or additions). Consider, now, the following scheme, Karatsuba's method:
Suppose we are working in binary, multiplying two $2n$-bit numbers u and v, where $u = u_{2n-1} \ldots u_1 u_0$ and $v = v_{2n-1} \ldots v_1 v_0$ are the binary digits. Set

$$\begin{aligned} u &= 2^n U_1 + U_0 \text{ and} \\ v &= 2^n V_1 + V_0, \text{ where} \\ U_1 &= u_{2n-1} \ldots u_n \text{ and} \\ U_0 &= u_{n-1} \ldots u_0 \end{aligned}$$

and similarly for V_i. Then

$$uv = (2^{2n} + 2^n) U_1 V_1 + 2^n (U_1 - U_0)(V_0 - V_1) + (2^n + 1) U_0 V_0.$$

Now this is just three multiplications of n-bit numbers, plus a few shifts and adds: we have reduce the number of multiplications needed from four to three. Write $T(n)$ for the time taken to multiply two n-bit numbers. Then for the old scheme, we have $T(n) = O(n^2)$. For the new scheme, $T(2n) \leq 3T(n) + cn$, where the cn is the multiplications and shifts. Applying the rule recursively, we get $T(2^k) \leq c(3^k + 2^k)$ (a different c) $= O(3^k)$, or, setting $k = \log_2 n$,

$$T(n) = O(3^{\log_2 n}) = O(n^{\log_2 3}) = O(n^{1.57}).$$

In the school scheme, doubling the length of the integers involved increases the time taken by a factor of 4. For the divide-and-conquer method it takes just 2.97 times as long. This might not seem much, but for large bignums it can make an immense difference. Ans, in practice, the cross-over point between the school method and Karatsuba is surprisingly low: perhaps at about a few tens of (decimal) digits.

This is by no means the best possible—asymptotically there exists methods that allow $T(n) = O(n^{1+\epsilon})$ for any fixed $\epsilon > 0$. Of course, the hidden constant gets huge as ϵ decreases.

There is also, more surprisingly, a method that employs Fourier transforms! This makes use of the convolution property of these transforms: long multiplication is just the convolution of the digits of the operands, and by means of a (discrete) Fourier transform, we map this into a space where we do simple digit against digit multiplication. Then the reverse transform takes us back to the answer. This is proved to be asymptotically a very good algorithm ($O(n \log n \log\log n)$ operations for n bit numbers), but the hidden constant makes this unused in most systems. See Knuth for details. There is some controversy over where the cross-over point between the Fourier/long multiplication methods is, some believe it is as low as 100-digit (base 10) numbers. If this is true, then the Fourier method really ought to be being used.

3.2.3 Division

Division is much harder. Again we use the school-algorithm of long division, but we have to explain how to "guess" the appropriate multiple to subtract at each stage: recall that to divide (say) 250 by 12, we guess that 2 is the largest quotient q with $12q \leq 25$. So 2 is the first digit of the answer. in general, we want to find the quotient of dividing a $n+1$ digit number by a n-digit number.

We find the answer in Knuth *The Art of Computer Programming* volume 2. Let us suppose we are working with digits base b, and are dividing $u = u_0 u_1 \ldots u_n$ by $v = v_1 v_2 \ldots v_n$, with $u/v < b$ (i.e. dividing a $n+1$-digit number by a n digit one). Set

$$\hat{q} = \min\left\{ \left\lfloor \frac{u_0 b + u_1}{v_1} \right\rfloor, b - 1 \right\}.$$

(the quotient can never be larger than $b - 1$.) Then we have two remarkable facts:
1. We have $\hat{q} \geq q$.
2. If $v_1 \geq \lfloor b/2 \rfloor$, then $\hat{q} - 2 \leq q \leq \hat{q}$, i.e. $\hat{q}$ is never more than 2 in error, independent of the base. The curious condition on v_1 is like a normalization, and can always easily be achieved by multiplying both u and v by $\lfloor b/(v_1 + 1) \rfloor$. (Proofs in Knuth).

So now we have an easy algorithm for division:
Start with non-negative integers $u_1 u_2 \ldots u_{n+m}$ and $v_1 v_2 \ldots v_n$, base b.

1. Set $d = \lfloor b/(v_1 + 1) \rfloor$, and scale u and v to $u_0 u_1 u_2 \ldots u_{n+m}$ and $v_1 v_2 \ldots v_n$, by d so that $v_1 \geq \lfloor b/2 \rfloor$ (notice that u gains a digit—this may be 0, though).
2. Set $j = 0$. This is the main loop variable.
3. Set $\hat{q} = \min\{\lfloor (u_j b + u_{j+1})/v_1 \rfloor, b - 1\}$.
4. Replace $u_j u_{j+1} \ldots u_{j+n}$ by itself minus $\hat{q}$ times $v_1 v_2 \ldots v_n$.

5. Set $q_j = \hat{q}$.
6. If the result of the subtraction is negative, replace q_j by $q_j - 1$, and add back $0v_1v_2\dots v_n$ to $u_ju_{j+1}\dots u_{j+n}$.
7. If it is still negative, increment q_j again, and add on another $0v_1v_2\dots v_n$.
8. Increment j, and loop if $j \le m$.

The quotient is $q_0q_1\dots q_m$, and the remainder is $u_{m+1}\dots u_{m+n}$ divided by d.

Knuth describes how to make an even *better* approximation for q that is at most 1 out, and rarely even that.

An example. Divide 111 by 19, base $b = 10$. Here $u_1u_2u_3 = 111$, $v_1v_2 = 19$, $n = 2$ and $m = 1$.

The scale factor $d = \lfloor 10/2 \rfloor = 5$. Thus we scale by 5, giving $u_0u_1u_2u_3 = 0555$, and $v_1v_2 = 95$.

First loop, $j = 0$. We find $\hat{q} = \min\{\lfloor 05/9 \rfloor, 9\} = \min\{0, 9\} = 0$. Subtracting $u_0u_1u_2 := u_0u_1u_2 - 0(v_1v_2)$, so nothing changes, and this is non-negative. Thus $q_0 = 0$.

Loop $j = 1$. Now $\hat{q} = \min\{\lfloor 55/9 \rfloor, 9\} = \min\{6, 9\} = 6$. So $u_1u_2u_3 := u_1u_2u_3 - 6(v_1v_2)$, i.e., $555 - 6 \cdot 95 = -15$. This is negative! Add one back: $-15 + 95 = 80$ is positive, so $q_1 = 5$.

This is the end of the algorithm. We have found that the quotient is $q_0q_1 = 05$, and the remainder is $80/d = 16$. A quick check: $5 * 19 + 16 = 95 + 16 = 111$, as required.

Note that $\min\{11/1, 9\} = 9$ as a guess for the first digit is way out from the true answer. The scaling by d is an essential part: otherwise the estimate $\hat{q}$ can be arbitrarily larger than q (up to the limit of the base b). Just imagine an unscaled division base 10^8: we might have to adjust $\hat{q}$ on the order of 100 million times!

3.2.4 GCDs

The Greatest Common Divisor of two non-negative integers is the largest integer that divides them both (or zero in the case that they both are zero). This is useful when we are considering rational numbers, viz numbers of the form n/m, with n and m both integers. To prevent hideous expression swell, we like to keep these in their lowest terms, i.e. divide out their common factor. And the GCD is interesting in its own right, too, of course. Complementary to the GCD is the Least Common Multiple, but one is easy to derive from the other using the formula $\gcd(u, v)\,\mathrm{lcm}(u, v) = uv$, for positive integers u and v.

The simplest way (to describe) of computing $\gcd(u, v)$ is as follows:
factorize u into primes $u = 2^{u_2}3^{u_3}5^{u_5}\cdots = \prod_p p^{u_p}$ and $v = \prod_p p^{v_p}$. Then

$$\gcd(u, v) = \prod_p p^{\min(u_p, v_p)}.$$

Needless to say, this algorithm is not perfect, as it requires the complete factorization of the integers involved, and current technology can only only cope with about 100-digit factorizations. However, there is a very old (300 BC) algorithm that bears the name of Euclid that does just what we want.

Euclid's algorithm:
Find the GCD of the non-negative integers u and v.

1. If $v = 0$ then return the value u.
2. Set $r = u \bmod v$, $u = v$, and $v = r$. Return to the first step.

We barely need an example:

$$\begin{aligned}
\gcd(1234, 4321) &= \gcd(4321, 1234) = \gcd(1234, 619) \\
&= \gcd(619, 615) = \gcd(615, 4) \\
&= \gcd(4, 3) = \gcd(3, 1) = \gcd(1, 0) = 1.
\end{aligned}$$

This algorithm terminates, as the non-negative integer v is decreased at each step (as is u at each step excepting possibly the first). A proof of the algorithm follows from the fact that

$$\gcd(u, v) = \gcd(v, u - qv)$$

for any integer q: any common divisor of u and v is also one of v and $u - qv$, and any common divisor of v and $u - qv$ must divide u and v. This terminates as $\min\{u, v\}$ decreases at each step (except possibly the first).

Knuth shows that this runs in average time $O(\log n)$, where u and v are bounded by n.

This algorithm is wonderful: it allows you to find the GCDs of huge numbers in very little time. It is the algorithm that most every algebra system implements, but, in fact, there are tweaks that can be made to improve it even further. For example, on binary machines we can take advantage of machine operations such as arithmetic shifts. We can make good use of the following properties of the GCD:

1. If u and v are both even, then $\gcd(u, v) = 2\gcd(u/2, v/2)$ (think of the prime factorizations).
2. If u is even and v odd, then $\gcd(u, v) = \gcd(u/2, v)$.
3. $\gcd(u, v) = gcd(u - v, v)$ ($q = 1$ in the above).
4. For u and v both odd, $u - v$ is even, and $|u - v| < \max(u, v)$.

And now we have enough to build an algorithm (Silver's method) that on uses binary shifts and simple subtractions:

Find the GCD of two positive integers u and v.

1. Set $k = 0$, and then repeat $k = k + 1$, $u = u/2$ and $v = v/2$ while both are even.
2. If v is even repeat $v = v/2$ until it is odd.
3. If u is even repeat $u = u/2$ until it is odd.
4. If $u \geq v$ then set $u = u - v$, else set $v = v - u$ (i.e. replace the larger of u and v by $|u - v|$).
5. If $u = 0$ return with the value $2^k v$, if $v = 0$ return with the value $2^k u$, else go back to step 2.

This time the example looks like:

$$\begin{aligned}
\gcd(1234, 4321) &= \gcd(617, 4321) = \gcd(617, 3704) \\
&= \gcd(617, 463) = \gcd(154, 463) \\
&= \gcd(77, 463) = \gcd(77, 386) \\
&= \gcd(77, 193) = \gcd(77, 116)
\end{aligned}$$

$$
\begin{aligned}
&= \gcd(77,29) = \gcd(48,29) \\
&= \gcd(3,29) = \gcd(3,26) \\
&= \gcd(3,13) = \gcd(3,10) \\
&= \gcd(3,5) = \gcd(3,2) \\
&= \gcd(3,1) = \gcd(2,1) \\
&= \gcd(1,1) = \gcd(0,1) = 1.
\end{aligned}
$$

In a real implementation we could stop as soon as we noticed the 1.

This algorithm terminates as $\max(u, v)$ is decreased at each step. It has the same asymptotic behaviour as Euclid's algorithm, but the increased number of steps necessary is easily made up for by their simplicity: only shifts and subtractions are needed, in comparison to the divisions necessary in Euclid's algorithm to find the remainder.

There is yet a third method to compute GCDs suitable for very large integers, due to Lehmer. This uses the same trick as the long division algorithm above: the leading few digits of the numbers make a good approximation to the behaviour to the whole numbers. If $u = u_1 u_2 \dots u_n$ and $v = v_1 v_2 \dots v_n$ are the digits of u and v, put $u' = u_1$, $v' = v_1 + 1$, $u'' = u_1 + 1$, and $v'' = v_1$. Then

$$u'/v' < u/v < u''/v''.$$

We carry out Euclid's algorithm on the single-precision pairs (u', v') and (u'', v'') until their sequence of quotients differs. Then the sequence of quotients for u and v themselves must be the same up to that point. If we keep track of the linear combinations of u and v we would have made in the normal algorithm we can do the whole corresponding sequence in one bignum computation.

This example is from Knuth:
Working base 10^5, compute $\gcd(27182818, 10000000)$.
We get $u' = 2718$, $v' = 1001$, $u'' = 2719$, and $v'' = 1000$. Then

u'	v'	q'	u''	v''	q''
2718	1001	2	2719	1000	2
1001	716	1	1000	719	1
716	285	2	719	281	2
285	146	1	281	157	1
146	139	1	157	124	1
139	7	19	124	33	3

At this point the quotients diverge, so we can go no further in single precision. Now the relevant modifications of u and v would have been

u	v	q
u	v	2
v	$u - 2v$	1
$u - 2v$	$-u + 3v$	2
$-u + 3v$	$3u - 8v$	1
$3u - 8v$	$-4u + 11v$	1
$-4u + 11v$	$7u - 19v$	?

Thus we can do a single computation for each of u and v:

$$u := -4u + 11v = (-4)27182818 + (11)10000000 = 1268728, \text{ and}$$

$$v := 7u - 19v = (7)27182818 - 19(10000000) = 279726.$$

And we can continue the algorithm in a similar manner.

There is an important extension to the concept of the GCD, namely that of the *cofactor*. Given integers u and v there exist cofactors u' and v' such that

$$uu' + vv' = \gcd(u, v).$$

This is particularly valuable in modular arithmetic, as we shall see later.

Euclid's algorithm extends easily to compute the cofactors in parallel with the GCD:
Given non-negative integers u and v, compute their GCD, and the cofactors.

1. Set the triples (vectors) $U = (1, 0, u)$ and $V = (0, 1, v)$.
2. If $V_3 = 0$, return the GCD U_3, and the cofactors $u' = U_1$ and $v' = U_2$.
3. Set $q = \lfloor U_3/V_3 \rfloor$.
4. Set $T = U - qV$, $U = V$, and $V = T$. Return to step 2.

At each step we have $uU_1 + vU_2 = U_3$, and $uV_1 + vV_2 = V_3$, so when we finish the required cofactors are U_1 and U_2.

Example: $\gcd(1234, 4321)$. We get $U = (1, 0, 1234)$, $V = (0, 1, 4321)$. Then
$U = (0, 1, 4321)$, $V = (1, 0, 1234)$
$U = (1, 0, 1234)$, $V = (-3, 1, 619)$
$U = (-3, 1, 619)$, $V = (4, -1, 615)$
$U = (4, -1, 615)$, $V = (-7, 2, 4)$
$U = (-7, 2, 4)$, $V = (1075, -307, 3)$
$U = (1075, -307, 3)$, $V = (-1082, 309, 1)$
$U = (-1082, 309, 1)$, $V = (4321, -1234, 0)$

So the GCD is 1, and the cofactors: $(1234)(-1082) + (4321)(309) = 1$.

Lehmer's algorithm generalises suitably, but Silver's is more tricky.

4 Polynomials

The next data object to consider is the *polynomial*. Polynomials can be easily represented in a computer program in many forms: lists, vectors, and so on, but unlike integers, the actual form of representation is of the utmost importance for efficient application in an algebra system. The choice of the correct representation can make orders of magnitude difference in the time it takes to run an algorithm.

4.1 Normality and Canonicality

Like all good computing courses we are going to start by considering how to represent our data.

First, though, we are going to define two important concepts which are valuable to keep in mind while choosing a representation: firstly, a scheme for representing polynomials (or indeed any class of arithmetic objects) is called *normal* if zero is uniquely represented in that scheme.

This is useful, as zero is the most important of all numbers, and if we can't recognize it when we have it, we will get into all sorts of problems. For example, if $p(x)$ were identically zero, but we didn't know this, we might accidentally divide by it, and deduce $1 = 0$. In a normal system we could not simultaneously have `x-x` and `0` as reduced forms. It would be strange, but possible, to have the former, but not the latter. It also allows us a test for equality: the question "is $f = g$?" is resolved by considering "is $f - g = 0$?" (Note that there are technical reasons why this may not be possible in general: see the section on Richardson's theorem later in the course.)

A scheme is called *canonical* if different representations always refer to different objects. Thus a canonical system will not allow us to have simultaneously the representations `x^2-1` and `(x-1)*(x+1)` for the polynomial $x^2 - 1$. This is strictly stronger than normality (and implies it), and lets us test for equality directly: two representations are of the same object if and only if they are identical in structure. Some regard canonicality as *too* strong: sometimes it is convenient to have more than one representation for a polynomial, e.g., keeping known factors explicit such as $(x - 1)$ in $(x - 1)(x^2 + 1)$.

4.2 Representation

There are many factors to be considered when deciding how to represent a polynomial in (say) Lisp, ranging from the ease of programming to the speed of the finished product, all these factors must be weighed against each other—rarely do we obtain a perfect result.

So to some specific representations, then. We might consider using strings: thus $x^2 - 1$ is represented as `"x^2-1"`. This is neither normal nor canonical as it stands, as we can also have the string `"(x-1)*(x+1)"`. Further, this form is difficult to manipulate (consider writing an arithmetic package!). However, it does have the advantage of being easy to read.

Another method is Polish or prefix notation: write in a list `(- (^ x 2) 1)` or maybe

```
(* (- x 1) (+ x 1))
```

Again this is not normal as it stands, but is slightly easier to program with (but not a lot).

This method can be made canonical if we work on it somewhat, by ordering terms and collecting like powers. There are many ways to do this, though, and we look at some of these ways in the following subsections.

4.2.1 Recursive and Distributed

In a *recursive* form, a multivariate polynomial is treated as a polynomial in one variable (the *main* variable), with coefficients polynomials in the other variables. Then the coefficients are themselves recursively polynomials one one variable over a lesser domain. This is equivalent to considering (say) $Q[x, y, z]$ as $Q[z][y][x]$, where x is the main variable. Thus $2x^2 + xyz + xy + z^2 = 2x^2 + ((z + 1)y)x + z^2$ is

```
(+ (+ (* (^ x 2) 2) (* (^ x 1) (* (^ y 1) (+ (^ z 1) 1)))) (^ z 2))
```

Each power of x occurs exactly once, and in each coefficient, each power of y happens just once, and so on. This method is canonical if require the powers of x (and then y, and then z) to be in decreasing order, but it is very hard to read.

In a distributed form the polynomial is represented as a sum of terms, each of which is a product of powers of the variables (and a constant). Generally the terms are ordered in a specific manner, e.g., lexicographic. This time we get

```
(+ (+ (+ (* (^ x 2) 2) (* (^ x 1) (* (^ y 1) (^ z 1))))
      (* (^ x 1) (^ y 1))) (^ z 2) )
```

which is $x^2+((z)y)x+(y)x+z^2$. Or, if we allow `*` and `+` to take many arguments, and writing `x` for `(^ x 1)` and the like,

```
(+ (* (^ x 2) 2) (* x y z) (* x y) (^ z 2))
```

Again this is canonical if we order the terms, and has the advantage of being a little easier to read.

4.2.2 Binary and N-ary operators

In the last case we noted that $+$ and $*$ can be generalised from their binary forms to N-ary forms: however, the binary form lends itself much better to a natural recursive implementation, rather than dispatching to list-traversing procedures.

4.2.3 Implicit and Explicit Operators

In the above examples we have explicitly included the operators `^`, `+` and `-`. This lets us read off easily what each term means, but is slightly tricky to code with: at each step we have a many-way choice depending on the operator. Also each operator uses up a bit of space which we can save by omitting them when we can guess them from the context. So we might write `(x 3)` for x^3 if we knew that this could be its only meaning in this position. To eliminate more than one operator we must pad degenerate cases, such as $3x^2+x+1$ would become

```
(((x 2) 3) ((x 1) 1) 1)
```

The polynomial is a list `(((var power) coeff) reductum)`: the `coeff` of course could be a polynomial in lesser variables. The `car` is the leading term, and the `cdr` is the *reductum*, a polynomial of lesser degree. In fact, we can go further than this, and use dotted-pairs to save more space.

```
(((x . 2) . 3) ((x . 1) . 1) . 1)
```

Here we know when we have got to the end of the polynomial—it is exactly when we have an atom.

The recursive form of $2x^2+xyz+xy+z^2$ is

```
(((x . 2) . 2) ((x . 1) ((y . 1) ((z . 1) . 1) . 1) . 0) ((z . 2) . 1) . 0)
```

and the distributed form is

```
(((x . 2) . 2) ((x . 1) (y . 1) (z . 1) . 1) ((x . 1) (y . 1) . 1)
 ((z . 2) . 1) . 0)
```

This is a list of terms, each of which is a list of variables to a power terminated a coefficient, terminated by a coefficient: `( ((var . pow) (var . pow) ... . coeff) ... . coeff)`.

4.2.4 Sparse and Dense

Each of the above representations have been *sparse*, which is to say that terms with 0 coefficient do not appear. Thus in $3x^3 + x + 1$ we do not put in a $0x^2$ term. On the other hand, a *dense* representation has a place-holder for *every* term, zero, or not. With a slight suspension of disbelief, we might consider a polynomial in three variables of degree at most n in each variable. This will require a fixed space of $O(3^{n+1})$ (remember the space for the degree zero terms!), in a distributed representation, which is most conveniently managed as a vector of the appropriate length. However, each term is uniquely specified by its index into the vector, so there is no need for operators or padding. Each slot contains just a ground-field coefficient.

Furthermore, arithmetic is very easy with this canonical representation: addition is trivial, multiplication only marginally harder, compared to the sparse representations, which require sorting of the cross-products as they are computed (which can be expensive).

A further advantage of a sparse representation is that is allows structure sharing: if we have the polynomial $x^2 + x + 1$, perhaps with representation `(((x . 2) . 1) ((x . 1) . 1) . 1)`, and then add $y + x^2$, represented by `(((y . 1) . 1) ((x . 2) . 1) . 0)`, we would get $y + 2x^2 + x + 1$, or

```
(((y . 1) . 1) ((x . 2) . 2) ((x . 1) . 1) . 1)
```

where the latter part, the `((x . 1) . 1) . 1` is in fact `eq` to the similar part in the first polynomial. This is a trivial example, but for very large polynomials, structure sharing is very important and lets us manipulate more data than it seems we have memory for!

4.2.5 Included and Extracted Variables

If we have a fixed number of variables (typically three for physicists, and four, or 11 or 23 or whatever for General Relativists) we can play the same trick as we did for operators, but this time eliminating the variables. Again this requires padding for missing variables, but we are left with lists of coefficients, which can be efficiently packed in a machine-dependent way. (This is the way that Camel works, and gains its speed: exponents packed in a single machine word can be added in parallel. It is the so-called "tabular form".) Now $2x^2 + xyz + xy + z^2$ looks like

```
((2 0 0 . 2) (1 1 1 . 1) (1 1 0 . 1) (0 0 2 . 1))
```

where we have a list of terms, `(expt expt expt . coeff)`, in the order x then y then z. The coding for this representation tends to be a little gungy, though.

4.2.6 Miscellaneous Bits

All the above have implicitly used a descending order of terms—this is probably because that reflects the way polynomials are normally written down, and the fact that most algorithms begin with the highest order term. However, it need not be like this, particularly for vector representations, which are equally easy to access in any order. Strangely, though, even division (when the result is exact) can be "worked from the other end." Just try it and see!

A more radical approach is to use a recursive dense representation: an explicit zero represents a whole subtree of zeros beneath it. This is a curious way of doing things—the polynomial $2x^2 + xyz + xy + z^2$ becomes

```
[ 2, [ [ 1, 1 ], 0 ], [ [ 1, 0, 0 ] ] ]
```

in the order x, y, then z, or even

```
(x . [ 2, (y . [ (z . [ 1, 1 ]) ]), (z . [ 1, 0, 0 ]) ])
```

with variables included.

Or even any mixture of the above: consider a representation which is dense in the first variable, the sparse in the rest. Then a polynomial like $x^2 + y^{100}$ might be `[ 1, 0, (((y . 100) . 1) . 0)]` if x was "first," but it would be `[1, 0, 0, ..., 0, (((x . 2) . 1) . 0)]` if y were first.

4.2.7 The Best One Is?

It all depends on the problem. We can make trades-off of space vs. time in moving from the tabular form (distributed, variables extracted) to the string representation. Perhaps on small machines, where space is a premium, we might choose the former, but on a large machine, where CPU crunching is the order of the day, we could want the latter. There are also trades-off in accessibility of terms in moving from the recursive sparse to the dense distributed. In some algorithms (e.g. division) we access terms in strict order, perhaps from largest degree to smallest degree, and in other algorithms we want to access the terms in somewhat random order. Integration is an example of the latter.

Also important is the order of the variables. In some problems (simultaneous polynomial equations) a change of the ordering of the variables can make an order of magnitude difference in the running time.

4.3 Example system

We are now going to study in a little detail of how REDUCE addresses the above choice. In fact REDUCE uses (at least) three different representations, namely the prefix (operators included), the sparse recursive, and the sparse distributed.

There is some advantage to doing this, in that we use the best (or perhaps, merely a good) representation for a particular job. We use prefix for input and output: it is convenient to translate the user's typed polynomial with pluses and hats directly into Polish notation. Then, for general arithmetic, we use the sparse recursive. This is useful for the basic arithmetic operations, as we tend to work on terms in specific order (usually decreasing). Other algorithms, of which integration is an important example, use sparse distributed. This is because the interaction between terms is (in some sense) limited, unlike in (say) division.

Some examples: if we type in `2x^2 + x*y*z + x*y + z^2`, this is first parsed to

```
(plus (times 2 (expt x 2)) (times x y z) (times x y) (expt z 2))
```

REDUCE can be somewhat verbose: it spells out the names of the operators. If we want to do something with this polynomial, like do some arithmetic on it, it is converted to

```
(((x . 2) . 2) ((x . 1) ((y . 1) ((z . 1) . 1) . 1)) ((z . 2) . 1) . 0)
```

(In fact, what *really* goes on is a lot more complicated, but that need not concern us now.)

In the integrator, when integrating with respect to x, this is viewed as the structure

```
(((x . 2) . 2) ((x . 1) . (((y . 1) ((z . 1) . 1) . 1))) . (((z . 2) . 1) . 0))
```

a mixed distributed and recursive form. The top level in `x` is distributed, but the coefficients are recursive. This is in keeping with the needs of integration, viz., one variable is important, the rest can be treated as constants.

(In fact, the two polynomials above have identical Lisp structure, it is the interpretation of the polynomial that is important here.)

The disadvantage of multiple representations, of course, is the time required to convert from one representation to another. However, if we only do this relatively infrequently (e.g., once on input), or the time taken is small compared to the savings made in having an appropriate representation, we win by doing so.

4.4 Polynomial Arithmetic

We now move on to the arithmetic of polynomials. The algorithms used depend very much on the data representation we have: we shall be using REDUCE's main representation, namely sparse recursive, variables included, operators extracted:

```
( ( ( var . exp ) . coeff ) . reductum )
```

where the reductum is another polynomial of the same type, or an integer, which is the trailing coefficient. We give names to parts of this structure:

Name	Structure	Extractor function	Lisp
leading power	`(var . exp)`	`lpow`	caar
leading term	`((var . exp) . coeff)`	`lterm`	car
leading coefficient	`coeff`	`lcoeff`	cdar
leading degree	`exp`	`ldeg`	cdaar
main variable	`var`	`mvar`	caaar
reductum	`reductum`	`red`	cdr

The coeff part, and the reductum are (in general) polynomials in "lesser" variables: recall that for canonicality we require a specific order on the variables.

A typical approach to addition of univariate polynomials is the following (written in Standard Lisp):

```
(de addf (u v)  % add two univariate standard forms, producing a standard form
  (cond ((zerop u) v)
        ((zerop v) u)
        ((atom u) (addd u v))
        ((atom v) (addd v u))
        ((greaterp (ldeg u) (ldeg v))
                  (cons (lterm u) (addf (red u) v)))
        ((greaterp (ldeg v) (ldeg u))
                  (cons (lterm v) (addf (red v) u)))
        (t (addt (plus (lcoeff u) (lcoeff v))
                 (lpow u)
                 (addf (red u) (red v))))))
```

In detail: if either of u or v is zero, we return the other (this is not strictly necessary—but it is a good trick, as we typically spend a lot of time adding zeros). If one of u or v is an integer (tested by `atom`) use `addd` to deal with it. If the degree of u is larger than that of v, then we just cons the leading term of u onto the sum of the reductum of u and v. And ditto the other way. Otherwise, the degrees of u and v are equal, and we make a new leading term using `addt`, which conses the sum of the old leading terms onto the sum of the reducta of u and v. For univariate polynomials, the coeffs are integers, so we can use `plus` (the Standard Lisp + function) to sum them.

Now there is a slight twist in the definition of `addt`: if the leading terms cancel, we do *not* want a leading zero:

```
(de addt (c p r)  % add the term c*p onto the standard form r
  (cond ((zerop c) r)     % they cancel
        (t (cons (cons p c) r))))
```

The definition of `addd` is

```
(de addd (u v)   % add an integer u to the standard form v, return a sf
  (cond ((atom v) (plus u v))
        ((zerop u) v)
        ((zerop v) u)    % redundant line
        (t (cons (lterm v) (addd u (red v))))))
```

If v is also an integer, add them directly, else recurse on the reductum.

For multivariate polynomials, there are a few significant changes: we must order the variables, and thus require a function to do so. We shall generalise a little, and define a function that orders two lpows, so, for example, we can use it uniformly over *all* lpows:

```
(de ordpp (p q)   % compare two lpows, return t if p is more important than q
  (cond ((orderp (car p) (car q)) t)
        ((orderp (car q) (car p)) nil)
        ((greaterp (cdr p) (cdr q)) t)
        (t nil)))
```

Detail: `orderp` is an intrinsic function that compares ids in some implementation dependent fashion. If the variable for p is greater than that for q, return true. If vice-versa, return false. Otherwise the two have the same variable, and we return true if the degree of p is greater than that of q; if not return false. Thus if $x > y$ we get $x^3 > x^2 > y^4$.

We can now write a multivariate version of `addf`:

```
(de addf (u v)        % add two standard forms, producing a standard form
  (cond ((zerop u) v)
        ((zerop v) u)
        ((atom u) (addd u v))
        ((atom v) (addd v u))
        ((ordpp (lpow u) (lpow v)
                (cons (lterm u) (addf (red u) v))))
        ((ordpp (lpow v) (lpow u)
                (cons (lterm u) (addf (red u) v))))
        (t (addt (addf (lcoeff u) (lcoeff v))
                 (lpow u)
                 (addf (red u) (red v))))))
```

Note: the use of `ordpp` in place of `greaterp`, and the recursive call on `addf` to add the coefficients. The functions `addd` and `addt` are the same as before.

Multiplication is just a little harder. One way is to use the identity

$$\begin{aligned} uv &= (\mathrm{lterm}(u) + \mathrm{red}(u))(\mathrm{lterm}(v) + \mathrm{red}(v)) \\ &= \mathrm{lterm}(u)\,\mathrm{lterm}(v) + \mathrm{lterm}(u)\,\mathrm{red}(v) + \mathrm{lterm}(v)\,\mathrm{red}(u) + \mathrm{red}(u)\,\mathrm{red}(v) \end{aligned}$$

The multiplication function `multf` would typically use `multt` to multiply terms, and `multtf` to multiply a term into a form. Each would involve recursive calls to `multf`.

The order of the terms is automatically correct if we use `addf` to sum the parts—this is equivalent to using an insertion sort on the terms as they are generated. A more efficient method would be to use, say, a quicksort on the terms, and this would decrease the time complexity from $O(n^3)$ to $O(n^2 \log n)$ to multiply two n-term polynomials. There are also the tricks we played with bignum arithmetic: there is a Karatsuba analogue, and a Fourier analogue.

Division over the integers is a whole new ball-game—to divide the polynomial u by the polynomial v we are asking for polynomials q and r (with integer coefficients) such that $u = qv + r$, such that $r = 0$ or $\deg r < \deg v$. At first glance, division of univariate polynomials is not too bad, but think about dividing $x^2 - 1$ by $2x + 2$ to give a quotient and remainder over the integers. We require a q and an r such that $q(2x + 2) + r = x^2 - 1$, with $\deg r < \deg(2x + 2) = 1$. Thus r is a constant, and by comparing leading coefficients, $2\,\mathrm{lcoeff}(q) = 1$, where $\mathrm{lcoeff}(q)$ is an integer. This is clearly impossible.

Multivariate are much worse: when we divide $2x - y$ by $x + y$ we get a quotient of 2 and a remainder $-3y$. On the other hand, dividing $-y + 2x$ by $y + x$, the quotient is -1, and the remainder is $3x$. In looking for a r with $\deg r < \deg v$ we must make a choice of variable for the degree comparison. And what do we mean by $(x + y)/2x$?

Let us start with a simple case, and suppose that we wish to divide in $Q[x]$, rather than $Z[x]$, i.e., we allow ourselves *rational* coefficients.

Divide the polynomial $u(x) = u_m x^m + \cdots + u_1 x + u_0$ by $v(x) = v_n x^n + \cdots + c_1 + c_0$, where $u_j, v_i \in Z$, $v_n \neq 0$, and $m \geq n \geq 0$ to give a quotient $q(x)$, and a remainder $r(x)$ over Q.

1. Set $k = m - n$.
2. Set $q_k = u_{n+k}/v_n$, and then $u_j = u_j - q_k v_{j-k}$ for $j = k$ to $n + k - 1$, and set $u_k = 0$.
3. Set $k = k - 1$, and if $k \geq 0$, return to step 2.
4. The quotient is $q(x) = q_{m-n} x^{m-n} + \cdots + q_0$, and the remainder $r(x)$ is (the new value of) $u(x)$.

This is really just the long division we are all used to, beefed up to look impressive. However, it does illustrate one point quite clearly: we only ever divide by v_n, and this happens exactly $m - n + 1$ times. Thus we have the so-called *pseudo*-division over Z:

$$v_n^{\delta+1} u = qv + r, \text{ with } \deg r < \deg u,$$

where $\delta = \deg u - \deg v$. It is important to realise that this is *not* true division, but just looks like it sometimes. It has some strange properties (e.g. consider $2x$ pseudo-divide $2x$), but is useful as it has most of the properties of division that we will require.

There are two ways to go about pseudo-division: firstly to multiply u by $v_n^{\delta+1}$ and use the normal division algorithm above in the knowledge that the division in step 2 will be exact. This will work, but has the slight disadvantage of expression swell: every term is magnified by $v_n^{\delta+1}$. The second route is to do the multiplication as we progress through the algorithm: this spreads the expression swell more thinly, and, incidentally, improves the algorithm so we have to do no (integer) divisions at all. This is the resulting algorithm:

Given $u(x)$ and $v(x)$, find $q(x)$ and $r(x)$ satisfying the above.

1. Set $k = m - n$
2. Set $q_k = u_{n+k} v_n^k$, then $u_j = v_n u_j - u_{n+k} v_{j-k}$ for $j = k$ to $n + k - 1$, and $u_j = v_n u_j$ for $j = 0$ to $k - 1$.
3. Set $k = k - 1$, and if $k \geq 0$, return to step 2.
4. The quotient is $q(x) = q_{m-n} x^{m-n} + \cdots + q_0$, and the remainder $r(x)$ is (the new value of) $u(x)$.

An example: divide $2x + 1$ into $x^3 + x + 1$.

$$
\begin{array}{ccc|cccc}
 & & & & 1 \times 2^2 & -1 \times 2 & 5 \\
\cline{4-7}
2 & 1 &) & 1 & 0 & 1 & 1 \\
 & & & 2 & 0 & 2 & 2 \\
 & & & 2 & 1 & & \\
\cline{4-5}
 & & & & -1 & 2 & 2 \\
 & & & & -2 & 4 & 4 \\
 & & & & -2 & -1 & \\
\cline{5-7}
 & & & & & 5 & 4 \\
 & & & & & 10 & 8 \\
 & & & & & 10 & 5 \\
\cline{6-7}
 & & & & & & 3
\end{array}
$$

The quotient is $q(x) = 4x^2 - 2x + 5$, and remainder $r(x) = 3$. As a check $qv + r = 8x^3 + 8x + 8 = 8u$.

4.4.1 Multivariate Polynomial Division

Having done pseudo-division over Z, we really need little more to generalize to division over multivariate rings: simply choose the "main variable" that we are going to work by, and treat every other variable as we did the integers in the above. Here is a simple example: divide $xy + y$ into $x^2 + y^2$, with x as our chosen variable.

$$
\begin{array}{ccc|ccc}
 & & & & 1 \times y^1 & -y \\
\cline{4-6}
\mathrm{y} & \mathrm{y} &) & 1 & 0 & y^2 \\
 & & & \mathrm{y} & 0 & y^3 \\
 & & & \mathrm{y} & \mathrm{y} & \\
\cline{4-5}
 & & & & -y & y^3 \\
 & & & & -y^2 & y^4 \\
 & & & & -y^2 & -y^2 \\
\cline{5-6}
 & & & & & y^4 + y^2
\end{array}
$$

The quotient is $xy - y$, and the remainder is $y^4 + y^2$. And check: $(xy + y)(xy - y) + y^4 + y^2 = x^2y^2 + y^4 = y^2(x^2 + y^2)$.

Similarly, dividing $(x + 1)y$ into $y^2 + x^2$ with y as the main variable we find a quotient of $(x + 1)y$ and a remainder of $x^2(x + 1)^2$.

4.5 Polynomial GCDs

4.5.1 Univariate

Let us return to the example of a little while back of computing a GCD:

$$
\begin{aligned}
u(x) &= x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5 \\
v(x) &= 3x^6 + 5x^4 - 4x^2 - 9x + 21
\end{aligned}
$$

Using the naïve Euclidean algorithm and using pseudo-division we get the following sequence of polynomial remainders:

$$
\begin{gathered}
-15x^4 + 3x^2 - 9 \\
15795x^2 + 30375x - 59535 \\
1254542875143750x - 1654608338437500 \\
12593338795500743100931141992187500
\end{gathered}
$$

(2 digits, 5 digits, 16 digits, 35 digits) Thus the polynomials are coprime, i.e. have no factor in common. We call such a sequence a *polynomial remainder sequence*, imaginatively enough.

Obviously, we could do much better by dividing the first remainder by 3 before continuing: this does not affect the sequence, apart from reducing the size of integer coefficients involved. We call the GCD of all the coefficients the *content* of the polynomial. The part left after dividing out the content is the *primitive part*: we cannot divide out any more and remain with integers. If we compute the primitive parts of the p.r.s. we get something like

$$
\begin{gathered}
5x^4 - x^2 + 3 \\
13x^2 + 25x - 49 \\
4663x - 6150 \\
1
\end{gathered}
$$

This is clearly a whole lot better. Can we do the same trick for multivariates? Well, consider the step of computing the content: this is itself several GCDs on the coefficients, which will themselves be polynomials. The recursion terminates as the coefficients are in fewer variables, but we are betting in doing this on several smaller GCDs being faster than the one big one.

In practice, we find that these *primitive* polynomial remainder sequences are not a win: we spend so much time trying to compute the primitive parts it is actually *slower* than the naïve method.

What we need is some way of *predicting* the content of each polynomial as it is generated (preferably with minimal computation), so we can divide it out immediately. Looking at the sequence of remainders, one sees (after a while) that the number you multiplied by to do the pseudo-division appears in the content *two* steps later, raised to some power. Hearn (the "inventor" of Reduce) used this in his "trial divisor" scheme. We know a factor that divides the content, so repeatedly try to divide it out until we can go no further. This method appears to work quite well with polynomials with integral coefficients, but slows up somewhat on more complicated domains where division is not so cheap (perhaps Gaussian integers).

Brown took this further and developed the *subresultant theorem*:
we wish to find the GCD of polynomials F_1 and F_2, using a remainder sequence $F_3 \ldots$. Let δ_i be the difference in degrees between F_i and F_{i+1}, and f_i the leading coefficient of F_i. Pseudo-division says that the remainder of dividing $f_i^{\delta_{i-1}+1} F_{i-1}$ by F_i is always integral. Call the remainder $\beta_{i+1} F_{i+1}$, where β_{i+1} is

the predictive part of the content to be determined (so the Euclidean algorithm has $\beta_{i+1} = 1$). So we get $\beta_{i+1} F_{i+1} = \text{remainder}(f_i^{\delta_{i-1}+1} F_{i-1}, F_i)$. If we set

$$\begin{aligned} \beta_3 &= (-1)^{\delta_1+1} \\ \beta_i &= -f_{i-2}\psi_i^{\delta_{i-2}} \qquad \text{where} \\ \psi_3 &= -1 \\ \psi_i &= (-f_{i-2})^{\delta_{i-3}} \psi_{i-1}^{1-\delta_{i-3}} \end{aligned}$$

then the F_i are all integral polynomials. Moreover, the coefficients grow only linearly in length at each step.

Using this the above example becomes

n	F	βF	f	δ	ψ	β
1	$x^8 + \cdots - 5$		1	2		
2	$3x^6 + \cdots + 21$		3	2	-1	-1
3	$15x^4 - 3x^2 + 9$	$-15x^4 + 3x^2 - 9$	15	2	-1	-1
4	$65x^2 + 125x - 245$	$-15795x^2 - 30375x + 59535$	65	1	-9	243
5	$9326x - 12300$	$-87431250x - 115312500$	9326		-25	-9375
6	260708	2863877380			-169	10985

where $\delta_1 = 2$, so $\beta_3 = -1$. Then $\beta_3 F_3 = \text{remainder}(F_1, F_2) = -15x^4 + 3x^2 - 9$, and so $F_3 = (-15x^4 + 3x^2 - 9)/(-1) = 15x^4 - 3x^2 + 9$. Also $\delta_2 = 2$ and $f_3 = 15$. The next step in the Euclidean algorithm is $\text{remainder}(F_2, 15x^4 - 3x^2 + 9) = -15795x^2 - 30375x + 59535$. Now, $\psi_4 = (-f_2)^{\delta_1}\psi_3^{1-\delta_1} = (-3)^2(-1)^{-1} = -9$, so that $\beta_4 = -f_2\psi_4^{\delta_2} = -3(-9)^2 = -3^5$. Hence $F_4 = (-15795x^2 - 30375x + 59535)/(-3^5) = 65x^2 + 125x - 245$. And so on.

This is not so good as the primitive p.r.s., but for multivariate GCDs we win enormously: there is no recursive computation of GCDs necessary. However, there is a little more we need for multivariate GCDs.

4.5.2 Multivariate

If we use Euclid's algorithm (or Brown's improvement) on the ring $Z[x_1, \ldots, x_n]$ we are essentially working in the ring $Z[x_1, \ldots, x_{n-1}][x_n]$ and this will only compute the dependence of the GCD on x_n. For example for

$$\begin{aligned} A(x_1, x_2) &= (x_1 - 1)x_2 + (x_1 - 1) \\ B(x_1, x_2) &= (x_1 - 1)x_2 + (-x_1 + 1) \end{aligned}$$

with x_2 as the main variable, we discover that A and B are relatively prime, whereas they clearly have the common factor $x_1 - 1$. To remedy this we use Gauss' Lemma:
Let p, $q \in R[x]$. Then $\text{cont}(pq) = \text{cont}(p)\,\text{cont}(q)$, and $\text{pp}(pq) = \text{pp}(p)\,\text{pp}(q)$. This allows the corollary

$$\begin{aligned} \text{cont}\,\text{gcd}(p, q) &= \text{gcd}(\text{cont}(p), \text{cont}(q)) \quad \text{and} \\ \text{pp}\,\text{gcd}(p, q) &= \text{gcd}(\text{pp}(p), \text{pp}(q)). \end{aligned}$$

And in particular, $\text{gcd}(p, q) = \text{gcd}(\text{cont}(p), \text{cont}(q))\,\text{gcd}(\text{pp}(p), \text{pp}(q))$.

This gives us a way of computing multivariate GCDs: simply multiply the GCDs of the contents by the GCDs of the primitive parts. (Note that this is *not* the same as computing a primitive PRS: think about the sequence of GCDs in each case).

Compute the GCD of the polynomials u and v over Z.

1. Set $u_1 = \mathrm{pp}(u)$, and $v_1 = \mathrm{pp}(v)$.
2. If $\deg v_1 = 0$ then return $\gcd(\mathrm{cont}(u), \mathrm{cont}(v)) \times \mathrm{pp}(u_1)$.
3. Set $r = \text{pseudo-remainder}(u_1, v_1)$, $u_1 = v_1$ and $v_1 = r$, and repeat from step 2.

(We halt when the degree of v_1 becomes zero). Here is a proof that this returns the GCD of the inputs u and v:
Let d be any divisor of both $\mathrm{cont}(u)$ and $\mathrm{cont}(v)$. Then d divides their GCD.

On the other hand, let d_1 be any primitive polynomial dividing both u and v. So d_1 divides u_1 and v_1. In the loop, $r = nu_1 - (\text{quotient})v_1$, for some n, so that d_1 divides r. Thus d_1 remains a divisor of u_1 and v_1. Hence at the end, d_1 must divide u_1 and therefore also $\mathrm{pp}(u_1)$.

From Gauss' relation above, we see when $d = \gcd(\mathrm{cont}(u), \mathrm{cont}(v))$ and $d' = \mathrm{pp}(u_1)$ that dd_1 must be exactly the GCD required.

4.6 Modular Methods

There is a problem with the above algorithms. After trying a few examples it becomes clear that the coefficients involved grow far too rapidly—the integer coefficients of univariates grow enormously, the degrees of the coefficients of multivariates increase embarrassingly, even using the subresultant algorithm. Even if the final result is quite small (the final GCD may be unity), the working you need to get there can be immense.

A general method of treating such algorithms is to do them modulo some integer:

$$\begin{array}{ccc} P & \stackrel{\text{big}}{\longrightarrow} & S \\ \downarrow & & \updownarrow \\ P_p & \stackrel{\text{small}}{\longrightarrow} & S_p \end{array}$$

take a problem P. Either we can generate a solution S directly using possibly large intermediate coefficients; *or* we can reduce the problem mod p, work the problem mod p with coefficients guaranteed to be smaller than p everywhere; we get a solution S_p. The hope is that S_p has some direct relationship (maybe is even equal) to S.

Think modulo a prime: Suppose u and v are polynomials over Z, and g is their GCD. So $u = gu'$ and $v = gv'$, say, and then $\bmod p$ we see $u_p = g_p u'_p$ and $v_p = g_p v'_p$. Hence g_p divides $\gcd_p(u_p, v_p)$. Note: it may not be the whole of the GCD $\bmod p$: consider the case of $u = x + 4$ and $v = x + 2 \bmod 2$. We have $g = 1$, so $g_p = 1$, but $u_p = x$ and $v_p = x$, so $\gcd_p(u_p, v_p) = x$. The GCD modulo p can be too large, but never too small.

An example: again let

$$\begin{aligned} u &= x^8 + x^6 - 3x^4 - 3x^3 + x^2 + 2x - 5 \\ v &= 3x^6 + 5x^4 - 4x^2 - 9x + 21 \end{aligned}$$

Modulo 5 this becomes:

$$\begin{aligned} u &= x^8 + x^6 - 3x^4 - 3x^3 + x^2 + 2x \\ v &= 3x^6 + x^2 + x + 1 \end{aligned}$$

Now, in $Z/5Z$, 3 has an inverse, namely 2. So $2v = x^6 + 2x^2 + 2x + 2$, and the remainder is $2x^2 + 3$, or $x^2 + 4$ when normalised. Successive remainders are x, and then 1 when normalised. Thus we see that the true GCD reduced mod 5 must divide 1. Thus the true GCD is of the form $5p(x) + 1$, for some $p(x)$. But considering the leading coefficients of u and v, we discover that $p(x) = 0$, i.e., the GCD is unity.

Thus we have: if there exists a prime p such that the GCD of u_p and v_p is unity, and p does not divide the leading coefficients of u and v, then u and v are coprime over Z.

That solves the coprime case, but what about when there is a non-trivial GCD? We are going to work along the lines of the following hope:
Suppose $p > 2|$ any coefficient in the GCD$|$ (the 2 is so that we can reconstruct the negative coefficients, too—we want to distinguish integers over the range $-p/2$ to $p/2$, viz., a range of size p), then perhaps $\gcd = \gcd_p$. So we compute mod p, and hope that the result is the true answer. Primes for which this fails are called "unlucky." For these primes the modular GCD is "too large."

$$\begin{array}{ccc} u, v & \longrightarrow & \gcd(u, v) \\ \downarrow & & \updownarrow \\ u_p, v_p & \longrightarrow & \gcd_p(u_p, v_p) \end{array} \quad = \text{ if } p \text{ large?}$$

Firstly, we must pick a p: how large does it need to be? It must be larger than any coefficient in the *answer*, which we do not yet know. So we have to guess. For polynomials $p(x) = p_n x^n + \ldots + p_1 + p_0$ and $q(x) = q_m x^m + \ldots + q_1 + q_0$ over Z such that q divides p we have the Landau-Mignotte inequality:

$$\sum_i |q_i| \le 2^m \left| \frac{q_m}{p_n} \right| \sqrt{\sum_j |p_j|^2}$$

and a corollary, as $|q_m| \le |p_n|$ (since the first divides the second)

$$|q_i| \le 2^m \sqrt{\sum_j |p_j|^2} \quad \text{for every } i.$$

And, in fact, $\operatorname{lcoeff}(\gcd(u, v))$ divides both $\operatorname{lcoeff}(u)$ and $\operatorname{lcoeff}(v)$, and therefore

$$\operatorname{lcoeff}(\gcd(u, v)) \,|\, \gcd(\operatorname{lcoeff}(u), \operatorname{lcoeff}(v)).$$

Thus we can sharpen the estimate. Note it is *not* true that the coefficients of the GCD are all smaller than those of the polynomials themselves: consider $\gcd(x^3 + x^2 - x - 1, x^4 + x^3 + x + 1) = x^2 + 2x + 1$. In general, they can be arbitrarily larger. This is in great contrast with the behaviour of the GCD of integers.

Thus we have a modular GCD algorithm:

Compute the GCD of the monic polynomials u and v.

1. Pick a random prime $p > 2\,\mathrm{Landau-Mignotte}(u, v)$.
2. Find $w = \gcd_p(u_p, v_p)$.
3. If w divides both u and v over Z, then return w. Else go back to step 1.

If, in step 3, w divides both u and v, it must be the true GCD: w cannot be too small, only too large, and if it is too large (i.e. it is larger than the true GCD) it will fail to divide one of u or v.

The algorithm works due to a theorem: "there exist only finitely many unlucky primes for a given problem." In practice almost all large primes are lucky, but the trick is to find one as small as possible, in order to get the most benefit from using smallnum arithmetic. If we are lucky, it has complexity $O(n^2(n + \log n + H))$, where $n = \max\{\deg u, \deg v\}$, and $H = \log_2 \max\{|u|, |v|\}$, where $|f|$ is the absolute value of the largest coefficient of the polynomial f. If we are unlucky, the complexity becomes $O(n^2(n + \log n + H)(n \log nH))$, where the last term is "the number of unlucky primes."

4.6.1 The Chinese Remainder Theorem

The above is called the "large prime method," as we pick a large prime to work with. Typically, however, this prime may be so large that we gain little or nothing, since the Landau-Mignotte bound is often very generous. It would be much better to use small primes if at all possible.

So apply a little number theory to the problem: suppose we want to determine an integer c, but we are only given $c \pmod{n}$ and $c \pmod{m}$. What can we say about c? Clearly we can only hope to determine c up to at most $\pmod{mn}$, and then only if n and m are coprime. In fact, this is exactly what we can do:

Chinese remainder theorem:
let n and m be coprime, and $a, b \in Z$. Then there exists a unique $c \pmod{mn}$ such that $c \equiv a \pmod{n}$, and $c \equiv b \pmod{m}$.

Proof:
By the extended Euclidean algorithm we can find integers λ and μ with $\lambda n + \mu m = 1$. Set $c = a + (b-a)\lambda n$. Then $\pmod{n}$, $c \equiv a$, and $\pmod{m}$, $c \equiv a + (b-a)(1-\mu m) \equiv a + b - a \equiv b$.

The principle of Chinese Remainder is directly applicable to (univariate) polynomials. If we have *polynomials* $a(x)$ and $b(x)$, then we can construct a polynomial $c(x) \pmod{mn}$ such that $c \equiv a \pmod{n}$ and $c \equiv b \pmod{m}$, for coprime integers n and m. To find $c(x)$, we simply apply the integer CR to each term in $a(x)$ and $b(x)$ independently. Thus if $c \equiv x+1 \pmod{3}$ and $c \equiv 2x+3 \pmod{5}$, we deduce $c \equiv 7x+13 \pmod{15}$.

The next plan is to use the CRT to build up a GCD bit by bit, using several small primes.

Modular algorithm, CRT version, u and v as before.

1. Set $bound = 2\,\mathrm{Landau-Mignotte}(u, v)$.
2. Set $primes\_so\_far = p$, where p is a random prime not dividing the leading coefficients of u and v.
3. Set $answer\_so\_far = \gcd_p(u_p, v_p)$.
4. If $answer\_so\_far = 1$, then return 1.
5. While $primes\_so\_far < bound$ do
6. Set p to be a new random prime, again avoiding the leading coefficients.
7. Set $w = \gcd_p(u_p, v_p)$.
8. If $w = 1$, return 1.
9. If $\deg w > \deg answer\_so\_far$ then loop: this means the current p is unlucky, and we should discard it.
10. If $\deg w < \deg answer\_so\_far$ then $answer\_so\_far = w$, $primes\_so\_far = p$, and loop: this means all the primes up to now were unlucky, and we have to start again.
11. Set $answer\_so\_far = \mathrm{Chinese\ Remainder}(answer\_so\_far, w)$, and $primes\_so\_far = primes\_so\_far \times p$.
12. (Test if $answer\_so\_far$ divides both u and v. If so $answer\_so\_far$ must be the true GCD; return it.)
13. Loop back to step 5.

14. Test that *answer_so_far* divides both u and v. If this fails, it means that *all* the primes up to now were unlucky, and we have to discard *answer_so_far* and start again from step 2. The chances of this happening, however, are exceedingly small. Else return *answer_so_far*.

We do the CR coefficient by coefficient on the polynomials in question. Again we rely on the fact that there are only finitely many unlucky primes. The complexity in this case is $O(n(n+H)^2)$.

A simple example: compute the GCD of $u = x^2 + 3x + 2$ and $v = x^2 + 5x + 6$. We find $\mathrm{LM}(u) = 2^2\sqrt{1^2+3^2+2^2} = 15$ and $\mathrm{LM}(v) = 2^2\sqrt{1^2+5^2+6^2} = 32$. Thus the bound will be 30.

p	u_p	v_p	$\gcd_p$	gcd sofar	p sofar
2	x^2+x	x^2+x	x^2+x	x^2+x	2
3	x^2-1	x^2-x	$x-1$	$p=2$ was unlucky	
				$x-1$	3
5	x^2-2x+2	x^2+1	$x+2$	$x+2$	15
7	x^2+3x+2	x^2-2x-1	$x+2$	$x+2$	105

In practice, we could stop when we get $x+2$, as we may identify it as the true GCD by trial division into u and v.

In the above we always assumed the polynomials were univariate—the CR method extends to cover multivariate polynomials, but there are some problems, of course. We used the Landau-Mignotte bound to estimate the size of the GCD: we need an analogue that will estimate the size (in the senses of magnitudes of coefficients, and degrees) of the coefficients, which are themselves polynomials. We mimic the method by working modulo irreducible polynomials in the ring of coefficients. For example in $Z[y][x]$, typical polynomials would be y, $y-1$, $y-2$, and so on. In fact this is equivalent to evaluating our polynomials at the values $y = 1, 2\ldots$ The CR step becomes an interpolation problem: we are given values of polynomials (in y, say) at certain points, and we need to reconstruct those polynomials from the values. There are the same problems as before, like unlucky "primes," and so forth.

over Z	over $Z[y]$
LM bound	degree bound in y
primes 2, 3, 5, ...	primes $\equiv$ irreducibles $y-1$, $y-2$, ...
reduction mod p	evaluation at $y=1$, $y=2$ ...
CRT	CRT $\equiv$ interpolation

4.6.2 Other Methods for the GCD

There are several other methods for computing GCDs of uni- and multivariate polynomials. A popular method is the "Hensel Method." This works by picking a single small prime p, and finds the GCD (mod p). Then, using a technique named "Hensel lifting," it successively deduces the GCD (mod p^2), then (mod p^3), and so on. After lifting sufficiently far (the Landau-Mignotte) bound, we must have the true GCD, or else we picked an unlucky prime.

A more unlikely method is the so-called "heuristic GCD," or "Z-adic" method. This substitutes a random integer n for the variable x in $u(x)$ and $v(x)$, computes the GCD of the resulting integers, then reconstructs a putative polynomial GCD by writing out the integer GCD base n, and replacing the powers of n by the corresponding powers of x. Thus: $\gcd(x^2+2x+1, x^2+3x+2)$ evaluated at 5, say, becomes $\gcd(36, 42) = 6$, and $6 = 5^1 + 1$, so the GCD is $x+1$. This is *not* an algorithm as it stands! However, computationally, it gets the answer right often enough to be worthwhile considering.

5 Matrices

5.1 Representation

We are now going to turn our attention briefly to matrices, and the manipulation thereof. First of all, how do we represent a matrix? The simplest way for a $m \times n$ matrix (m rows, n columns) would be to have a vector of m elements, each of which is a vector of n elements containing the values. This canonical representation is a dense one, where access is through a simple double-index. Being dense, this is most suitable for representing *dense* matrices, i.e., those with most elements non-zero. For a *sparse* matrix (most entries are zero), another representation might be more efficient. An example could be a list of pairs of coordinates and value:

```
( ((1 . 2) . 3) ((50 . 23) . -11) ((100 . 23) . 34) )
```

this is a 100 by 100 matrix (or a 100 by 50, or ...) that has three non-zero entries of 3, -11, and 34. Ensuring zero terms never appear, and ordering the terms makes this canonical. Arithmetic is a trifle harder with this representation: for the dense representation addition is just a double loop, and multiplication is a triple loop. However, we must consider all terms, zero or non-zero, making addition $O(n^2)$. In this sparse case for addition we only consider non-zero terms, but we must do a linear search for matching terms. This makes it have order only the square of the number of *non-zero* terms. A more sophisticated technique might employ a hash table in place of the search.

There are dozens of other sparse representations, each taking advantage of particular aspects of the matrices in question, such as is the matrix symmetric or anti-symmetric (we need only store half of it), or triangular, or banded, and so on. There is a huge literature from numerical analysis on just this question.

A more recent idea (1988), is to recursively partition the matrix into quadrants, and use a tree representation. Whenever an entire subtree is zero, we replace the tree by a zero. Thus

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 & 0 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 3 & 4 & 0 & 0 \\ 0 & 0 & 5 & 6 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & 4 & 5 & 6 \end{pmatrix}$$

becomes

```
[ [ [ 1, 2, 3, 0 ], 0, [ 4, 5, 7, 8 ], [ 6, 0, 9, 0 ] ],
  0,
  [ 0, [ 1, 2, 5, 6 ], 0, 0 ],
  [ [ 3, 4, 7, 8 ], 0, [ 9, 1, 0, 4 ], [ 2, 3, 5, 6 ] ] ]
```

taking the quadrants in the order top left, top right, bottom left, and bottom right. This appears to be a good representation, in that it represents both sparse and dense matrices reasonably efficiently. The code for arithmetic is slightly tricky, but is nothing impossible.

5.2 Arithmetic

Addition is not worth spending much time over, except that we should note in passing that the efficiency depends very much on the type of representation used.

For multiplication, we have the straightforward algorithm: to multiply U into V, take the inner product of each row of U with each column of V. This takes n^3 multiplications to find the product of two n^2 matrices. Of course, there are better ways of doing things. Strassen gave the following method: let U and V be 2 by 2 matrices over an arbitrary ring (not necessarily commutative), and we wish to compute $W = UV$. First find the values

$$\begin{aligned}
m_1 &= (u_{12} - u_{22})(v_{21} + v_{22}) \\
m_2 &= (u_{11} + u_{22})(v_{11} - v_{22}) \\
m_3 &= (u_{11} - u_{21})(v_{11} + v_{12}) \\
m_4 &= (u_{11} + u_{12})v_{22} \\
m_5 &= u_{11}(v_{12} - v_{22}) \\
m_6 &= u_{22}(v_{21} - v_{11}) \\
m_7 &= (u_{21} + u_{22})v_{11}
\end{aligned}$$

And then

$$\begin{aligned}
w_{11} &= m_1 + m_2 - m_4 + m_6 \\
w_{12} &= m_4 + m_5 \\
w_{21} &= m_6 + m_7 \\
w_{22} &= m_2 - m_3 + m_5 - m_7
\end{aligned}$$

So now let U and V be general n by n matrices. We divide each into submatrices

$$U = \begin{pmatrix} U_1 & U_2 \\ U_3 & U_4 \end{pmatrix} \qquad V = \begin{pmatrix} V_1 & V_2 \\ V_3 & V_4 \end{pmatrix},$$

and then use the above relations to compute $W = UV$, recursively multiplying the submatrices.

Notice that this uses just 7 multiplications (plus some more additions). Hence, by an inductive argument as before, two matrices can be multiplied in just $O(n^{\log_2 7}) = O(n^{2.8})$ operations. There are other, better (in the sense of time complexity) algorithms, but the cross over point between them and the naïve algorithm becomes rapidly so enormous as to be useless (in the practical sense).

5.3 Matrix Determinant and Inverse

Now we shall look at square matrix inversion and determinant. Clearly these two operations are related, but it is possible to compute one without knowing the other. Firstly, we consider the determinant.

For an n by n matrix there is a specific formula to compute the determinant, namely the sum of signed products of terms. Unfortunately, this involves $n!$ terms, and is unwieldy to compute with. We might try to expand by minors: this is fine if we choose the right minors, but we get intermediate expression swell—the intermediate workings tend to be much larger than the final result, as the minors cancel each other out in very curious ways. There is the method used in numerical analysis: the determinant of a triangular matrix is the product of the elements on the diagonal. We can convert any matrix to triangular form by using Gaussian elimination (subtracting a rational multiple of one row from another is a unimodular transformation). This is fine for floating point entries, but a disaster for algebraic entries, as we must produce canonical rational forms at every point, meaning an absurd number of GCDs to keep the forms in lowest terms. Also, for a matrix of polynomials it is a waste of time introducing rational forms to compute a quantity which is ultimately a polynomial. There are a purely polynomial forms of Gaussian elimination (in the sense they do not introduce any divisions) that are akin to a GCD computation, but they, too, suffers from expensive intermediate expression swell. The Bareiss algorithm (see DST) is a good example to consider.

A much better method is to work modularly: either a large prime or a CR variant can be made to work. Now to apply this, we must have a bound on the possible size of the determinant. To this end we have Hadamard's bound:
For an integer matrix U, we have

$$|\det(U)|^2 \leq \prod_i \left(\sum_j |u_{ij}|^2 \right).$$

(This is the volume of a n-dimensional box of sides the rows of U.) Now there is no analogue of unlucky primes: if the matrix is singular (mod p), then p divides the determinant. Thus, for the large prime variant, we simply work (mod p), and the result is the true determinant. Unfortunately, Hadamard's bound is often very generous, and we gain little, if at all by working modulo a prime bigger than twice it. We get better results from the CR:

Find the determinant of the integer matrix U.

1. Set $bound = 2 \times \text{Hadamard}(U)$.
2. Pick a random prime p, set $primes\_so\_far = p$, and $det\_so\_far = \det_p(U_p)$.
3. Pick a random prime p distinct from any previous.
4. Set $primes\_so\_far = p \times primes\_so\_far$.
5. Set $det\_so\_far = CR(det\_so\_far, \det_p(U_p))$.
6. If $primes\_so\_far > bound$, then return det_so_far, else go back to step 3.

This has running time $O(n^4)$, but in practice it works very well—the largest number that appears in the algorithm is the final answer. This method adapts well to polynomial entries, too.

5.4 Matrix Inversion

We can adapt some of the above mentioned algorithms to find the determinant: particularly the Gaussian elimination method. Once we have a triangular matrix it only requires simple back-substitution to compute the inverse. This, of course, suffers exactly the same problems of expression swell as the determinant computation. A better approach is to adapt the modular method.

However, we now come up with unlucky primes again: if the determinant is divisible by p, then the matrix is singular (mod p), and is thus non-invertible. But it is very easy to spot this quickly, and move on to another prime. Are more important problem is that the inverse may have non-integral entries, and the modular methods seem only suitable for integer results. One solution might be firstly to compute the determinant (by a modular method, of course!), and then compute the matrix $\det(U)U^{-1}$, which has integral entries.

There is also a way of reconstructing rational numbers from modular information directly, though. Let m be the modulus, and suppose $|a|$, $|b| < \sqrt{m/2}$, with b coprime to m. From the extended Euclidean algorithm, we can find λ and μ with $\lambda b + \mu m = 1$, which means $\lambda b \equiv 1 \pmod{m}$. In other words, $a/b \equiv \lambda a$. Further, this modular number (i.e., λa) corresponds to a unique rational number in the range 0 to $\sqrt{m/2}$, for suppose $a'/b' \equiv \lambda a$, where $|a'|, |b'| < \sqrt{m/2}$, then $ab' \equiv a'b$. But $|ab'|, |a'b| < m/2$, so $ab' = a'b$, or $a/b = a'/b'$.

Reversing this, i.e., computing a rational from a modular number, is yet another application of the extended Euclidean algorithm. Suppose we wish to find the rational corresponding to $n \pmod{m}$. Apply Euclid's algorithm to n and m, to give a decreasing sequence of integers a_i, where $a_i = b_i n + c_i m$. This means we have $n \equiv a_i/b_i$ at each step. Then for the first element a_i in the series with $a_i < \sqrt{m/2}$, we have the required rational a_i/b_i.

An example: work (mod 9), and find the rational which is represented by 4 (mod 9). Note that $\sqrt{9/2} = 2.12\ldots$, so if such a rational exists, it will have numerator and denominator no larger in magnitude than 2. The remainder sequence is

$$(9, 1, 0), \quad (4, 0, 1), \quad (1, 1, -2), \text{and } (0, -4, 9).$$

Now the third member of this sequence says that $1 = 1(9) + (-2)4$, which means $4 \equiv -1/2 \pmod{9}$.

6 User Algorithms

In this section we are going to move away from the underlying representations a little, and concentrate more on descriptions of algorithms. We shall be covering three topics (out of very many), namely polynomial factorization, algebraic approximation, and Taylor series. These are called "user algorithms," as they are typical algorithms that might be used in the solution of larger problems.

6.1 Repeated Approximation

We will see later on in the course about solution of differential and algebraic equations and the like, but we are going to look at *approximate* solutions first. This usage of the word "approximate" in this context means a Taylor series approximation: we never demean ourselves to consider mucking about with real or complex numbers, every solution we get is algebraically correct up to the requisite order. Formally we want to solve the equation $f(x, y) = 0$ in the form $y = F(x) + O(x^n)$ where x is assumed small. First we find a zero-order approximation y_0 by solving $f(0, y_0) = 0$ for y_0. Next, suppose y_n is a nth order solution, i.e., $f(x, y_n) = O(x^{n+1})$. Putting $y = y_n + \eta$, we solve $f(x, y_n + \eta) = 0$ to order $n+1$ (neglecting terms of order $n+2$ and higher), to produce η, the correction term. Then $y_{n+1} = y_n + \eta$ to order $n+1$.

To describe the algebraic approximation method, we are going to work through an example, namely

$$y^2 = 1 + x.$$

Firstly we find a zero-order solution: $y_0 = 1$. Next, suppose y_n is a solution to order n, and put $y = y_n + \eta$. So $(y_n + \eta)^2 = 1 + x$, or $y_n^2 + 2y_n\eta + \eta^2 = 1 + x$. But η is of order $n+1$, so η^2 can be ignored. Similarly, we have $y_n\eta = y_0\eta$ to the same order. Thus

$$y_n^2 + 2y_0\eta = 1 + x$$

or

$$\eta = (1 + x - y_n^2)/2y_0.$$

So, in this case, we have an iterative formula,

$$\begin{aligned} y_0 &= 1 \\ y_{n+1} &= [y_n + (1 + x - y_n^2)/2]_{n+1} \end{aligned}$$

(the square brackets mean "to order $n+1$"). A few terms are $y_1 = [1 + (1 + x - 1^2)/2]_1 = 1 + x/2$, $y_2 = [1 + x/2 + (1 + x - (1 + x/2)^2)/2]_2 = 1 + x/2 - x^2/8$, and so on in the usual Taylor expansion of $\sqrt{1+x}$.

The good thing about this method is that it also applies equally well to differential equations—consider, for example

$$y' + y = \epsilon y^2.$$

with $y(0) = 1$, and ϵ small. It is a *perturbed* equation, that is, it is nearly linear, and we employ the near-linearity to find an approximate solution.

We look for a solution of the form

$$y = y_0 + \epsilon y_1 + \epsilon^2 y_2 + \cdots$$

For the zeroth order solution, $y = y_0$ we have

$$y_0' + y_0 = 0, \qquad \text{to } O(\epsilon)$$

so $y_0 = e^{-x}$.

Next, $y = y_0 + \epsilon y_1$, to $O(\epsilon^2)$. Substituting into the original equation,

$$\begin{aligned} (y_0 + \epsilon y_1)' + (y_0 + \epsilon y_1) &= (y_0' + y_0) + \epsilon(y_1' + y_1) \\ &= \epsilon(y_1' + y_1) \qquad \text{as } y_0' + y_0 = 0 \\ &= \epsilon(y_0 + \epsilon y_1)^2 \\ &= \epsilon y_0^2 + 2\epsilon^2 y_0 y_1 + \epsilon^3 y_1^2 \\ &= \epsilon y_0^2 \qquad \text{to } O(\epsilon^2) \\ &= \epsilon e^{-2x}. \end{aligned}$$

Thus: $y_1' + y_1 = e^{-2x}$, to $O(\epsilon)$. Rather than solve this, we solve the more general

$$z' + z = e^{-nx}.$$

This turns out to have $z = Ae^{-x} + e^{-nx}/(1-n)$ as the general solution, when $n \neq 1$, and A is an arbitrary constant.

Thus, we see $y_1 = Ae^{-x} - e^{-2x}$. From the initial condition $y_1(0) = 0$ (as $y(0) = 1$ and $y_0(0) = 1$) we get

$$y_1 = e^{-x} - e^{-2x},$$

and then

$$y = e^{-x} + \epsilon(e^{-x} - e^{-2x}) \qquad \text{to } O(\epsilon^2).$$

We can take this further and find the next term, $y = y_0 + \epsilon y_1 + \epsilon^2 y_2$ to $O(\epsilon^3)$. This time,

$$\begin{aligned} y' + y &= (y_0' + y_0) + \epsilon(y_1' + y_1) + \epsilon^2(y_2' + y_2) \\ &= 0 + \epsilon e^{-2x} + \epsilon^2(y_2' + y_2) \\ &= \epsilon(y_0 + \epsilon y_1 + \epsilon^2 y_2)^2 \\ &= \epsilon y_0^2 + 2\epsilon^2 y_0 y_1 \qquad \text{to } O(\epsilon^3) \\ &= \epsilon e^{-2x} + 2\epsilon^2 e^{-x}(e^{-x} - e^{-2x}), \end{aligned}$$

so that

$$y_2' + y_2 = 2e^{-2x} - 2e^{-3x} \qquad \text{to } O(\epsilon).$$

Using the equation in z above, this is simply solved as

$$y_2 = -2e^{-2x} + e^{-3x} + e^{-x}.$$

The solution for y this far is

$$\begin{aligned} y &= e^{-x} + \epsilon(e^{-x} - e^{-2x}) + \epsilon^2(e^{-x} - 2e^{-2x} + e^{-3x}) \\ &\qquad \text{to } O(\epsilon^3). \end{aligned}$$

For working this sort of problem in Reduce, it is useful to know about the use of `LET`:

```
let x^8 = 0;
```

means that all powers 8 or higher of `x` are to be ignored, i.e. work to $O(x^8)$.

6.2 Taylor Series

Leading on from the series approximation method of the previous section we see that useful thing to be able to do is to find the Taylor series expansion of a function $f(x)$ about a point. This enables us to discern useful information about f, such as orders of zeros, limits and so on. Clearly we cannot hope to compute the *entire* power series when it is infinite, so we must generally truncate to some finite power. This is exactly what some implementations of a Taylor package do: you choose a degree *in advance*, and compute a power series, discarding all terms higher than the degree. Using elementary analysis

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \dots,$$

where we are expanding about a. If a happens to be a removable singularity (such as for $\sin x/x$ about 0), the derivatives can be computed using l'Hôpital's rule. So to (say) multiply two such series we do the obvious cross-multiplication of terms, throwing away any terms that exceed our required degree.

This is fine for a one-off calculation of a series, but what if we now wish to find a few more terms, say in a complicated expression composed of several series? This means we will have to re-compute *all* the series from scratch, this time with a larger degree, and then re-compute the expression using the new series. This is a problem, particularly when we do not know the required degree in advance: consider the limit

$$\lim_{x \to 0} \frac{\sin \tan x - \tan \sin x}{\arcsin \arctan x - \arctan x \arcsin x}.$$

We can find this by computing Taylor series expansions for the numerator and denominator, and comparing the orders of the zeros at 0—but how far do we need to go?

The method we are going to describe, devised by A. Norman, gives the impression we are manipulating *full* power series: in fact it is a lazy evaluation method, using iterative rules to compute terms only when they are needed.

This method has been implemented in Reduce. To use, the package must be loaded by

```
load_package tps;
```

and then to find a series try

```
ps(sin x/x,x,0);
```

to expand $\sin x/x$ about $x = 0$. The default number of terms shown is given by `psexplim()`, and `psexplim 10;` will increase it to 10, and so on.

6.2.1 Norman's Method

A power series is going to be represented as an expression that involves the index i, which evaluates to the coefficient of x^i in the power series. Thus the power series for x might be $(i = 1)$, where, following Knuth, the notation $(\mathit{conditional})$ takes the value 1 when the conditional is true, and zero otherwise. The representation for e^x would be $1/i!$. In general the expression would be implemented as a *memo function*, i.e. a procedure that saves a value once computed (e.g. on a property list), so the next time it is required, no computation is necessary.

Addition of two power series is achieved by adding the expressions. So adding the power series x to e^x, will result in

$$(i = 1) + 1/i!.$$

Again, in practice, the memos can be merged.

Differentiation and integration is easy, the derivative of *expr* being

$$(i + 1) \times \mathit{expr}[i \rightarrow i + 1]$$

where $\mathit{expr}[a \rightarrow b]$ means "replace a by b in the expression *expr*." Integration is just as easy:

$$(1/i) \times \mathit{expr}[i \rightarrow i - 1]$$

The integral of x will be $(1/i)(i - 1 = 1)$, or, equivalently $(i = 2)/2$, which is just $x^2/2$. When we come to multiplication and division things become a little harder. The product of expr_1 and expr_2 is

$$\sum_{j=0}^{i} \mathit{expr}_1[i \rightarrow j]\mathit{expr}_2[i \rightarrow i - j].$$

This is just the normal cross-product of terms. Note that i is free, and j is bound in the above.

For division of expr_1 by expr_2, we get

$$\begin{aligned} \mathit{expr}_3 &= (i = 0)\mathit{expr}_1[i \rightarrow 0]/\mathit{expr}_2[i \rightarrow 0] \\ &\quad + (i \neq 0)\frac{\mathit{expr}_1 - \sum_{j=0}^{i} \mathit{expr}_3[i \rightarrow i - j]\mathit{expr}_2[i \rightarrow j]}{\mathit{expr}_2[i \rightarrow 0]}, \end{aligned}$$

where expr_3 is the quotient form. Note this is recursively defined. (If the denominator is 0 at 0, the formula is just a little more complicated.)

Now we have a reasonable representation, we can turn to the generation of power series of general functions. Again we lead by example. Suppose we want to expand $y = 1/(1 - x)$ about 0. Differentiating,

$$\begin{aligned} y' &= 1/(1 - x)^2 = y^2 \\ &\text{or} \\ y &= \int y^2. \end{aligned}$$

Looking at this in series terms, if y_i is the coefficient of x^i, this last equality says that

$$y_i = (1/i) \sum_{j=0}^{i-1} y_j y_{i-j-1}$$

with $y_0 = y(0) = 1$. This is because the coefficient of x^i in y^2 is just $\sum_{j=0}^{i} y_j y_{i-j}$, and the rule for integration replaces i by $i-1$ and divides by i.

This is a iterative formula to compute the y_i—we find, of course, that $y_i = 1$, for each i that we compute. We need only compute as many terms as we want, and it is easy to compute more once we have stopped. And, further, we can employ this expression as a part of a more complex series expansion, such as $1/(1-x)^2$, for example.

Here's a harder example: compute the series for $y = \sec x$. This time, $y' = \tan x \sec x$, so

$$y_i = (1/i) \sum_{j=0}^{i-1} z_j y_{i-j-1},$$

and $y_0 = 1$, where z_i is the coefficient of x^i in $z = \tan x$. However, $z' = 1 + \tan^2 x = 1 + z^2$, so we see that from $z_0 = 0$, and because 1 has series $(i = 0)$, and z^2 has series $\sum_{j=0}^{i} z_j z_{i-j}$, so

$$z_i = (1/i)\big((i-1=0) + \sum_{j=0}^{i-1} z_j z_{i-j-1}\big).$$

Piecing this all back together, we find we have a well-founded iterative method of computing the y_i. Thus:

$$\begin{array}{ll} z_1 = (1/1)(1 + z_0^2) = 1 & z_2 = (1/2)(z_0 z_1 + z_1 z_0) = 0 \\ z_3 = (1/3)(z_0 z_2 + z_1^2 + z_2 z_0) = 1/3 & z_4 = (1/4)(z_0 z_3 + z_1 z_2 + z_2 z_1 + z_3 z_0) = 0 \end{array}$$

and hence

$$\begin{array}{ll} y_1 = (1/1)(z_0 y_0) = 0 & y_2 = (1/2)(z_0 y_1 + z_1 y_0) = 1/2 \\ y_3 = (1/3)(z_0 y_2 + z_1 y_1 + z_2 y_0) = 0 & y_4 = (1/4)(z_0 y_3 + z_1 y_2 + z_2 y_1 + z_3 y_0) = 5/24 \end{array}$$

And so on. In practice, this method is very practical and efficient, as long as we use memos to cut out re-computation.

6.3 Polynomial Factorization

Given an integer, there is a straightforward (if slow) method for finding its irreducible factors—just consider each integer in turn up to the square root. If we want to factorize a polynomial, though, there is no immediate analogous procedure, and we must look to more sophisticated ideas. The object of the exercise is to factorize an arbitrary polynomial f over Z into a product $f = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ where the p_i are distinct irreducible polynomials. Gauss' lemma tells us we need only consider factors that are polynomials over Z themselves: if a polynomial factorizes at all, we can find a factorization over Z.

There have been algorithms to factorize polynomials around for a very long time—the following is attributed to Kronecker, and this relies on the observation that a factorization of a polynomial is reflected in a factorization of the integer we get from substituting a value into the polynomial.

We proceed by finding factors of degree 1, then degree 2, and so on. Let $f(x)$ be the polynomial over Z we wish to factorize. Suppose that $ax + b$ is a factor of f, so that $f(n)/(an + b)$ is an integer, for all integral n.

Thus to factorize f, we pick a pair of suitable n, 0 and 1, say, interpolate a linear polynomial though each of the integer divisors of $f(0)$ and $f(1)$ to compute the corresponding values of a and b, and then test to see if $ax+b$ is a factor of f. Note we must consider *all* divisors, prime or not. It is easiest to describe this using an example:

Factorize $f = x^3+1$ over $Z[x]$. $f(0) = 1$, and this has divisors ± 1. $f(1) = 2$, with divisors ± 2, ± 1. A linear polynomial $an+b$ must take one of the following pairs of values at the points $n = 0$ and $n = 1$:
-1,-2; -1,-1; -1,1; -1,2; 1,-2; 1,-1; 1,1; 1,2.

The polynomials to try are
$-x-1$; -1; $2x-1$; $3x-1$; $-3x+1$, $-2x+1$, 1, $x+1$.

We find that $x+1$ is the unique (up to scalar multiplication) non-trivial factor, and the quotient is x^2-x+1. This must be irreducible, as it has no degree 1 factors (by trial division of the factors above). Hence the complete factorization of f is

$$x^3+1 = (x+1)(x^2-x+1).$$

To find a degree two factor, we would have to take three points, ± 1 and 0, say, and consider all the combinations of all factors of the values of f at those points, interpolate quadratics through them, and test them for divisibility. To find a cubic factor, we look at *four* points, and so on.

Clearly this is not a suitable algorithm to find factors of polynomials with degrees larger than 2, or, at a pinch, 3. It is, however, quite efficient at finding linear factors, particularly if we are prepared to spend some time in choosing evaluation points (other than just using 0 and 1) to make the values have small factorizations—or better still, are prime. (This is also a suitable method for programming on a micro, due to its simplicity.) For degree 5 or more this method totally infeasible, and we must look to more sophisticated methods.

The first simplifying assumption that we can make is to assume f is monic, for say $f(x) = 2x^3+x+1$. Then instead of $f(x)$ use $2^2 f(x) = (2x)^3 + 2(2x) + 4 = g(2x)$, where $g(x) = x^3+2x+4$. Then g will have exactly the same shape of factorization as f, and we may freely convert between the two. This does introduce some expression swell, though.

6.3.1 Square-Free Factorization

We can get quite a long way to factorization by use of simple GCD computations in the case that f has repeated factors: e.g. $f = (x+1)^2(x^2+1)$ has the repeated factor $x+1$. We say the factor $(x+1)$ has multiplicity 2 in f; the factor (x^2+1) has multiplicity 1.

Let f be an arbitrary polynomial over Z, and suppose we can factorize f into linear factors over C:

$$f = \prod_i (x-a_i)^{n_i},$$

where the $a_i \in C$ (algebra hackers note we need only work in a finite extension of Q). Now take the derivative of f

$$f' = \sum_i \left(n_i (x-a_i)^{n_i-1} \prod_{j \neq i} (x-a_j)^{n_j} \right),$$

Thus, for each i, $(x-a_i)^{n_i-1}$ divides both f and f', and what is more $(x-a_i)^{n_i}$ does *not* divide f' (it divides all the terms except one). Thus the GCD of f and f' is exactly

$$\gcd(f, f') = \prod_i (x-a_i)^{n_i-1}.$$

Note that the RHS is in fact a polynomial over Z, since the LHS is. Now we get $f/\gcd(f, f') = \prod_i(x - a_i)$. Write g for this product, $g = \prod_i(x - a_i)$. Then

$$\gcd(g, \gcd(f, f')) = \prod_{n_i>1}(x - a_i),$$

and then

$$\frac{g}{\gcd(g, \gcd(f, f'))} = \prod_{n_i=1}(x - a_i).$$

The LHS of this equation is defined over Z, as all divisions are exact, hence the RHS, which is the product of all terms of multiplicity one in f, must be a polynomial over Z.

We can repeat the above on $\gcd(f, f')$, which contains just those factors of f of multiplicity of 2 or more, but with their multiplicity reduced by one. This extracts the product of those terms of multiplicity exactly two. Then repeating we find the factors of multiplicity three, and so on.

Doing this We have factorized f into the form $f = \prod f_i^i$, where the f_i are square-free and relatively prime. This is the *square-free decomposition* of f.

An example: $f = x^6 + 6x^5 + 14x^4 + 18x^3 + 17x^2 + 12x + 4$ Then $f' = 6x^5 + 30x^4 + 56x^3 + 54x^2 + 34x + 12$. So $\gcd(f, f') = x^2+3x+2$, and $g = f/\gcd(f, f') = x^4+3x^3+3x^2+3x+2$. Next, $\gcd(g, \gcd(f, f')) = x^2+3x+2$, and $g/\gcd(g, \gcd(f, f')) = x^2 + 1$.

Thus the product of all terms of multiplicity one is $x^2 + 1$, and $x^2 + 3x + 2$ is the product of all terms of multiplicity two or more, reduced by one. Thus, $f = (x^2 + 1)(x^2 + 3x + 2)^2$. Notice that $x^2 + 3x + 2$ is not irreducible, but factors as $(x + 1)(x + 2)$. This means we will have to work harder on the square-free parts individually to get their factors.

6.3.2 Berlekamp's Algorithm

Now we have a product of polynomials, each of which is square-free, and so we need some way of separating those factors of equal multiplicity: square-free factorization on $x^2 - 1$ reveals that all its factors are of multiplicity one, but it does not say whether it is a product of one degree two polynomial, or two degree one polynomials. As with the GCD and matrix arithmetic, we look to modular methods.

There exist several modular techniques to factorize polynomials, but they all have features in common. In particular, they proceed by first computing a factorization (mod p), and then try to reconstruct the "true factorization" (i.e., the factorization over the integers) from the modular one. Berlekamp's is one such method.

Let $f(x)$ be our polynomial over Z, and suppose for the nonce, that we know how to factorize polynomials over Z/pZ for prime p. Now if we can find a prime p with $f_p(x)$ irreducible, we know that f must be irreducible over Z, as any factorization of f over Z will project down to one over Z/pZ. Now there are many "if"s in this, not the least being that there are polynomials that factorize (mod p), for every p, but which are irreducible over Z. The favourite example of this is $x^4 + 1$.

For

$$\begin{array}{ll} p \equiv 1 \pmod 8 & \prod(x \pm (1 \pm \sqrt{-1})/\sqrt{2}) \\ p \equiv 3 \pmod 8 & (x^2 - \sqrt{-2}x - 1)(x^2 + \sqrt{-2}x - 1) \\ p \equiv 5 \pmod 8 & (x^2 - \sqrt{-1})(x^2 + \sqrt{-1}) \\ p \equiv 7 \pmod 8 & (x^2 - \sqrt{2}x + 1)(x^2 + \sqrt{2}x + 1) \\ p = 2 & (x + 1)^4 \end{array}$$

This is just one example of an infinite set (the Swinnerton-Dyer polynomials) that demonstrate this behaviour.

First of all, we must ensure f is square-free (mod p). The process of square-free factorization described before follows through (mod p) just as before, but with one additional twist: we can have a non-constant f with $f' = 0$. For example, $(x^p + 1)' = 0$ (mod p). This happens if, and only if, $f(x)$ can be written in the form $f(x) = g(x^p)$, for some g (as every coefficient disappears, we must have multiplied them by p). So we can continue the process using the factorization $f(x) = g(x^p) = g(x)^p$. The last equality is a consequence of Fermat's little theorem: for each integer a (mod p), $a^p = a$ (mod p) (see DST for proofs).

Thus we may now assume f is square-free.

Berlekamp's algorithm itself also relies on Fermat's little theorem. Since $a^p = a$ (mod p), for every a (mod p), it follow immediately that every integer (mod p) is a root of the polynomial $x^p - x$, and therefore

$$x^p - x = (x-0)(x-1)(x-2)\dots(x-(p-1)) \pmod p.$$

This is enough ammunition to factorize a modular polynomial, as follows. Suppose $f(x)$ factorizes into irreducible polynomials

$$f = f_1 f_2 \dots f_r,$$

where we do not yet know r. We ensured f was square-free, so the f_i are relatively prime. Let s_1, s_2, ..., s_r be integers (mod p). By the CRT, there is a polynomial v such that

$$v = s_i \pmod{p, f_i}, \tag{1}$$

where we work both modulo p and the polynomials f_i. The degree of v is less than the degree of the product of the f_i, namely the degree of f.

Now if $s_i \neq s_j$, then $\gcd(f, v - s_i)$ is divisible by f_i, but not by f_j. Thus we can decompose f. Putting the relations together,

$$v(x)^p = s_j^p = s_j = v(x) \pmod{p, f_j},$$

we conclude

$$v(x)^p = v(x) \pmod{p, f}, \tag{2}$$

but substituting $v(x)$ for x in the above,

$$v(x)^p - v(x) = (v(x)-0)(v(x)-1)\dots(v(x)-(p-1)) \pmod p. \tag{3}$$

Thus if v satisfies (2), then f divides the LHS of (3), so each f_i must divide one of the factors on the RHS of (3). Hence $v(x) =$ an integer (mod p, f_i), i.e., v satisfies (1) for some s_i. This proves that the solutions of (1) are exactly the solutions of (2).

So we have now the problem of finding solutions to (2). But this, surprisingly, is a *linear* problem, thanks to Fermat's little theorem. To solve this, consider the Q-matrix

$$\begin{pmatrix} q_{00} & q_{01} & \cdots & q_{0,n-1} \\ q_{10} & q_{11} & \cdots & q_{1,n-1} \\ \vdots & \vdots & & \vdots \\ q_{n-1,0} & q_{n-1,1} & \cdots & q_{n-1,n-1} \end{pmatrix},$$

where $x^{pk} = q_{k,n-1}x^{n-1} + \cdots + q_{k1}x + q_{k0}$ (mod $p, f(x)$). Considering as polynomial as a vector of its coefficients, (post-)multiplication by Q corresponds to the computation of the p^{th} power of that polynomial. i.e., if the vector $\mathbf{v}$ corresponds to v, then $\mathbf{v}Q$ corresponds to v^p. Thus the solutions of (2) are the solutions of $\mathbf{v}Q = \mathbf{v}$, i.e., eigenvectors of Q (mod p) for the eigenvalue 1.

Berlekamp's algorithm. Given f, square-free (mod p), find its irreducible factors.

1. Compute the matrix Q.
2. Find the eigenvectors of Q for the eigenvalue 1. One such is always $(1, 0, \ldots, 0)$, since integers are always solutions to (2). The dimension of the eigenspace is the number of irreducible factors of f.
3. Pick an eigenvector v, and compute $\gcd(f, v - s)$ for each integer $s \pmod p$. If this fails to decompose f completely, choose another eigenvector and repeat.

An example: $p = 5$, $f = x^3 + 2$. All computations will be (mod 5). Firstly, f is square-free, as $\gcd(f, f') = \gcd(x^3 + 2, 2x^2) = 1$. We get

$$Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 2 & 0 \end{pmatrix},$$

as $x^p = x^5 = 3x^2$, $x^{2p} = x^{10} = 2x$. To compute the eigenspace for the eigenvalue 1, we consider the matrix

$$Q - 1I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 2 & 0 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 4 & 3 \\ 0 & 2 & 4 \end{pmatrix}.$$

This has dimension 2, i.e., there are two irreducible factors of f. Picking the eigenvector $(0, 2, 1)$, corresponding to the polynomial $v = 2x + x^2$:

$$\begin{aligned} \gcd(f, v - 0) &= \gcd(x^3 + 2, 2x + x^2) = 1 \\ \gcd(f, v - 1) &= x^2 + 2x + 4 \\ \gcd(f, v - 2) &= 1 \\ \gcd(f, v - 3) &= x + 3 \\ \gcd(f, v - 4) &= 1 \end{aligned}$$

Hence $x^3 + 2 = (x^2 + 2x + 4)(x + 3)$.

This turns out to be a very good algorithm, running in time $O(n^3 + prn)$. This is fine for small p, but there exist variants (see Knuth) for large p that run in time like $\log^3 p$.

The computation of eigenvalues may be simply performed: we want to solve $vM = 0$ for some v. We can multiply on the right by an invertible matrix to get $vB = 0$, where B is lower triangular—this can be obtained by unimodular column operations on M. The above matrix reduces to

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 0 \end{pmatrix}.$$

So if $(v_1, v_2, v_3)B = 0$, we see that $v_3 =$ any, $v_2 + 3v_3 = 0$, and $v_1 =$ any. Picking $v_3 = -1$, and $v_1 = 0$, so $v_2 = 3$. Thus an eigenvector is $(0, 3, -1)$, or equivalently, $(0, 2, 1)$.

6.3.3 Factorization over Z

We have solved the problem of factorization over the modular field Z/pZ. Can we use this information to derive a factorization over Z? As noted above there are some technical problems with this: a polynomial may factorize (mod p), for every p, but still be irreducible.

How about the CRT? This works much as we might expect, but there is a problem here, too. Suppose we have a factorization of f (mod p) and (mod q): $f = \prod g_i^{e_i} \pmod p$, and $f = \prod h_j^{k_j} \pmod q$. To apply the

CRT we must determine which g_i corresponds to which h_j. We can match up the degrees of the g_i and h_j, but in general this is not going to be enough, and we are going to have to try *all* suitable combinations. This is a combinatorial explosion, and will be very expensive, of course.

We can glean some information from the degrees of the modular factors, using the fact that modular images never underfactor. For example, $f = x^5 + x + 3$ factorizes mod 2 into $(x^2 + x + 1)(x^3 + x^2 + 1)$. This says that the factorization over the integers might be a 2, 3 split or a 5 split (i.e., irreducible). This is compatible with the mod 3 factorization $x(x^2 + x + 2)(x^2 + 2x + 2)$, which implies possible splits of 1, 2, 2 or 1, 4 or 2, 3 or 5. However, mod 13 we find the factorization $(x + 8)(x^4 + 5x^3 + 12x^2 + 8x + 2)$, with splits 1, 4 and 5, which cannot be compatible with any mod 2 factorization apart from the single, irreducible 5 split (it *is* compatible with the mod 3 factorization, though). Thus f must be irreducible. Unlucky primes are not so unlucky in this case!

6.3.4 Hensel's method

If the problem is relating several primes, how about working modulo a single prime? Hensel devised a method that, given a factorization $\pmod p$, we deduce a factorization $\pmod{p^2}$, and then one $\pmod{p^3}$, and so on until we have moved to a large enough power of p that we are assured of the correct coefficients. So we need a bound on the coefficients of the factors—but this is just the Landau-Mignotte bound again

$$|\text{factors of } f| \le 2^{\deg f}\sqrt{\sum |f_i|^2}.$$

As before, polynomials can behave in strange ways: factors can have coefficients that are unexpectedly large. For example,

$$\begin{gathered} x^{41} - x^{40} - x^{39} + x^{36} + x^{35} - x^{33} + x^{32} - x^{30} - x^{27} + x^{23} + x^{22} - x^{21} - x^{20} + x^{19} + x^{18} - x^{14} \\ -x^{11} + x^9 - x^8 + x^6 + x^5 - x^2 - x + 1 \end{gathered}$$

has a factor with coefficient 5774. Furthermore, a slight perturbation of this,

$$\begin{gathered} x^{41} - x^{40} - x^{39} + x^{36} + x^{35} - x^{33} + x^{32} - x^{30} - x^{27} + x^{23} + x^{22} - x^{21} - x^{20} + x^{19} + x^{18} - x^{14} \\ -x^{11} + x^9 - x^8 + x^6 + x^5 - x^2 - x + 1 \\ +(x-1)^8 \\ = \\ x^{41} - x^{40} - x^{39} + x^{36} + x^{35} - x^{33} + x^{32} - x^{30} - x^{27} + x^{23} + x^{22} - x^{21} - x^{20} + x^{19} + x^{18} - x^{14} \\ -x^{11} + x^9 - 8x^7 + 29x^6 - 55x^5 + 70x^4 - 56x^3 + 27x^2 - 9x + 2 \end{gathered}$$

has irreducible factor

$$\begin{gathered} x^{33} + 7x^{32} + 27x^{31} + 76x^{30} + 174x^{29} + 343x^{28} \\ +603x^{27} + 968x^{26} + 1442x^{25} + 2016x^{24} + 2667x^{23} \\ +3359x^22 + 4046x^{21} + 4677x^{20} + 5202x^{19} + 5578x^{18} \\ +5774x^{17} + 5774x^{16} + 5578x^{15} + 5202x^{14} + 4677x^{13} \\ +4046x^{12} + 3359x^{11} + 2667x^{10} + 2016x^9 + 1442x^8 + 968x^7 \\ +603x^6 + 343x^5 + 174x^4 + 76x^3 + 27x^2 + 7x + 2 \end{gathered}$$

which has coefficients 80 times larger.

So now suppose we have a factorization $\pmod p$,

$$f = g_1 h_1 \pmod p,$$

(where for simplicity we have just two factors) and suppose further that g_1 and h_1 are coprime $\pmod p$ (which we can ensure by insisting f is square-free). Thus, from extended Euclid over the modular field, there are G and H such that

$$Gg_1 + Hh_1 = 1 \pmod p.$$

We want to "lift" this to a factorization $\pmod{p^2}$, so let us look at this equality in a little more detail, in particular $\pmod{p^2}$:

$$f = (g_1 + pg_2)(h_1 + ph_2) \pmod{p^2}$$

for some g_2 and h_2. Then

$$f = g_1 h_1 + p(g_1 h_2 + h_1 g_2) \pmod{p^2},$$

or

$$\frac{f - g_1 h_1}{p} = g_1 h_2 + h_1 g_2 \pmod p.$$

Comparing this with the Euclidean equality above, we see that

$$g_2 = \frac{f - g_1 h_1}{p} H \qquad h_2 = \frac{f - g_1 h_1}{p} G.$$

Thus using these values for g_2 and h_2, we have obtained a factorization of $f \pmod{p^2}$. Next we want p^3:

$$f = (g_1 + pg_2 + p^2 g_3)(h_1 + ph_2 + p^2 h_3) \pmod{p^3},$$

with the g_3, h_3 to be determined. As before, we get

$$\frac{f - g_1 h_1 - p(g_1 h_2 + h_1 g_2) - p^2 g_2 h_2}{p^2} = g_1 h_3 + h_1 g_3 \pmod p,$$

or

$$g_3 = HR \qquad h_3 = GR, \pmod p$$

where $R = (f - (g_1 + pg_2)(h_1 + ph_2))/p^2$ is the remainder after subtracting the "answer so far."

In general, given a factorization of accuracy p^n, we get one of accuracy p^{n+1} in just the same way.

An example: $f = x^3 + 6x^2 + 2x + 12 = (x+1)(x^2+2) \pmod 5$. The factors are coprime, and

$$(3x+2)(x+1) + (2)(x^2+2) = 1.$$

Now

$$R = (f - (x+1)(x^2+2))/5 = x^2 + 2,$$

then

$$g_2 = (2)(x^2+2) = 2x^2 + 4 = 1 \pmod{5, 3x+2},$$

and

$$h_2 = (3x+2)(x^2+2) = 3x^3 + 2x^2 + 6x + 4 = 0 \pmod{5, 2}.$$

Thus

$$f = (x + 1 + 5(1))(x^2 + 2 + 5(0)) = (x+6)(x^2+2) \pmod{5^2}.$$

In fact $f = (x+6)(x^2+2)$ is the true factorization.

So the algorithm for factorizing over the integers is as follows: pick a good prime p (i.e., not unlucky), factorize $\pmod p$, and then Hensel lift until we get $p^n > 2\,\text{Landau} - \text{Mignotte}(f)$. Now we have either the true factors, or we started with a poor prime for which f over-factors (e.g. a Swinnerton-Dyer polynomial). We discover whether we have the true factors by the simple test of trial divisions of the putative factors into f. If we have an over-factorization, the only recourse we have now is to take first pairs, then triples, and so on, of modular factors, multiply them together $\pmod{p^n}$, and test if these combinations are factors. (We are re-combining the over-factored parts). In the end, we either get true factors, or we discover f was irreducible when all combinations fail.

In the above, we have always ensured that the leading coefficient of f was always unity, and showed how to make a substitution to make this happen. On the other hand, we can work directly with the original polynomial to avoid the coefficient growth, but this introduces problems with the leading coefficient. In the factorization $6x^2+5x+1 = (2x+1)(3x+1)$ we must find some way of determining that the leading coefficient is distributed in the way it is (the factorization over a modular field destroys all such information).

One way of doing this is to make f monic (mod p) at the outset (by multiplying by the inverse of the leading coefficient), and use the fact that if g divides f, then cg divides $a_n f$, where the coefficient c is chosen such that the leading coefficient of cg is a_n, viz., $cg_m = a_n$, or $c = g_m^{-1} a_n$. So we modify the factorization algorithm as follows:

1. factorize (mod p) the monic version of f_p.
2. multiply the Landau-Mignotte bound by a_n, as we must now bound the coefficients of $a_n g$.
3. in the Hensel algorithm, use $a_n^{-1} f$ rather than f (computed modulo the appropriate power of p).
4. division tests of the sort "does g divide f?" are replaced by tests like "does $a_n g \pmod{p^k}$ divide $a_n f$?"

An alternative approach is to consider f/a_n modulo some p, and use the fraction reconstruction method (as we did for matrix inverses).

6.3.5 Notes on Factorization

There are a few other factorization algorithms, an example being the Cantor-Zassenhaus algorithm (see Knuth). This factorizes (mod p) using a probabilistic method that extracts about half of the factors of f in an iteration.

Another, less likely way, is the Z-adic method. Here we substitute a large integer n for x in $f(x)$, determine all the divisors of $f(n)$ over the integers, and try to reconstruct a factor of $f(x)$ by writing the divisors base n, and putting x for n. This does work, but involves huge integers: n must be at least twice as large as the coefficients of the factors, thus $n \approx 2.2^{\deg f}\sqrt{\sum |f_i|^2}$, which means

$$f(n) \approx n^{\deg f} \approx \left(2^{\deg f+1}\sqrt{\sum |f_i|^2}\right)^{\deg f}$$

using the Landau-Mignotte inequality. This number is huge.

There is a variant of Hensel's algorithm which proceeds from p to p^2, to p^4 to p^8, and so, doubling the p-adic accuracy at each step. This is fine, and it reaches the Landau-Mignotte bound, very rapidly, but it also has a tendency to overshoot: consider $p = 3$, and a bound of 26. We get 3, 9, then 81, as opposed to 3, 9, then 27 for the linear version. Overshoot means we are working modulo integers much larger than we need. A hybrid version moves quadratically at first, and then creeps up linearly when we get close.

Multivariate factorizations are achieved in more or less the obvious way: reduction by a prime $y-a$ is the same as evaluation at $y=a$, and Hensel lifting happens in an analogous manner. The Landau-Mignotte bound analogue is unnecessary, as the degree of a factor must be less than the polynomial.

7 Advanced Algorithms I—Polynomial Elimination

A great deal is known about the solution of linear simultaneous equations: the theory has been around for a large number of years, and many alternative algorithms have been presented for their solution. However, relatively little is known nowadays about the solution of higher degree simultaneous equations. For example, solve the set $\{x^2+y^2-2, xy-1\}$ for x and y. There are the usual ad-hoc methods of trying to eliminate variables and back-substitution, but these often fail on problems with polynomials of degrees higher than 4, for the usual Galois-theoretic reasons.

Elimination theory used be a rave amongst the Victorians, and many mighty tomes have been written on the subject, most of which have been rightfully forgotten. One technique has survived, though, that of using resultants.

7.1 The Resultant

The Victorians loved playing with polynomials, and solving sets of multivariate simultaneous equations. One way of doing this was to take two polynomials in x and y, say, and eliminate x from them to produce a new (larger) polynomial in y alone. One way to do this was invented by Sylvester: let $f(x)$ and $g(x)$ be polynomials over some ring R (e.g. Z[y]). Suppose $f=\sum_{i=0}^n f_i x^i$, and $g=\sum_{j=0}^m g_j x^j$. The Sylvester matrix of f and g is

$$\begin{pmatrix}
f_n & f_{n-1} & \dots & f_1 & f_0 & 0 & 0 & \dots & 0 \\
0 & f_n & f_{n-1} & \dots & f_1 & f_0 & 0 & \dots & 0 \\
\dots & & & & & & & & \dots \\
0 & \dots & 0 & f_n & f_{n-1} & \dots & f_1 & f_0 & 0 \\
0 & \dots & 0 & 0 & f_n & f_{n-1} & \dots & f_1 & f_0 \\
g_m & g_{m-1} & \dots & g_1 & g_0 & 0 & 0 & \dots & 0 \\
0 & g_m & g_{m-1} & \dots & g_1 & g_0 & 0 & \dots & 0 \\
\dots & & & & & & & & \dots \\
0 & \dots & 0 & g_m & g_{m-1} & \dots & g_1 & g_0 & 0 \\
0 & \dots & 0 & 0 & g_m & g_{m-1} & \dots & g_1 & g_0
\end{pmatrix}.$$

There are m lines of f_i's and n lines of g_j's. The *resultant*, $\text{resultant}_x(f,g)$, is the determinant of the above matrix. It is a polynomial in the coefficients of f and g, and as a consequence, if the coefficients of f and g are themselves polynomials in another variable, y, say, the resultant is simply a polynomial in y.

There are better ways of computing the resultant, particularly the *sub-resultant* algorithm of Collins, which we saw earlier applied to the computation of GCDs. The resultant has a close connection with the GCD, and the GCD always divides the resultant.

Sylvester used his matrix for solving simultaneous multivariate polynomial equations: the resultant eliminates a variable from a pair of equations, and you are able to work solutions of the resultant back to solutions of the original equations. It is not a very efficient solution, however, because the degree of the determinant is generally much larger than the degrees of the original polynomials. Like the GCD, we can find polynomials F and G with $Ff+Gg=r$, so if x_0 is a value of x such that $f(x_0)=g(x_0)=0$, we see that necessarily $r=F(x_0)f(x_0)+G(x_0)g(x_0)=0$.

This gives us a way to compute the zeros of general, multivariate $\{f, g\}$. Find their resultant and equate it to 0. This gives us a new equation, but in one less variable (viz., the one we eliminated via the resultant). Solving this, we can backtrack and check that the root we found is indeed a root of the original set. For larger sets of polynomials, we can eliminate from polynomials in pairs.

An example: solve $\{x^2 + y^2 - 2, xy - 1\}$ for x and y. We find $\text{resultant}_x(x^2 + y^2 - 2, xy - 1) = (y^2 - 1)^2$. Thus we want to solve $(y^2 - 1)^2 = 0$. This is easy to do, and we get $y = \pm 1$. Then, the complete solutions are $(x, y) = (1, 1)$, and $(-1, -1)$.

This is fine for small examples, but it has (at least) two important problems: firstly that of *parasitic solutions.* Not every solution of the resultant relates back to a solution of the original polynomials; the resultant can introduce additional values which merely serve to confuse. For example, consider the equations

$$(y-1)x + (y-1) \qquad \text{and} \qquad (y+1)x.$$

Eliminating x from the second equation, we get

$$(y-1)x + (y-1) \qquad \text{and} \qquad y^2 - 1$$

which have as root $(x, y) = (42, 1)$ (amongst many others), which is not a solution to the original set of equations.

The second, and more disastrous, problem is that of expression swell. The degree of a resultant is typically $(\deg_x f + \deg_x g)\max\{\text{degrees of polynomials in the coefficients}\}$. This is to say that the resultant polynomials grow in size rapidly. Similarly their integer coefficients grow at an alarming rate. For even a small set of polynomials, in low degree in a few variables, we are soon forced to manipulate large degree polynomials with large coefficients.

In this chapter we look at two new ways of solving multivariate polynomial equations, namely using Gröbner bases (using Buchberger's algorithm), and using Wu's algorithm.

7.2 Gröbner Bases

First we look at the Buchberger algorithm, which computes an object called a *Gröbner base.* This has come very much into vogue as of late, mainly due to its extrmely strong mathematical properties. We start with a finite set of polynomials $F = \{f, g, h \ldots\}$ of which we wish to find the simultaneous roots. To do this we compute an equivalent set of polynomials, called a Gröbner base. This set is equivalent in the sense that every root of F is a root of the Gröbner basis, and vice versa. However, the Gröbner basis is typically very much easier to solve; for example, it is often in triangular form, i.e., there is a polynomial in (say) x alone, then a polynomial in x and y alone, and so on. This allows us to compute the roots by simple univariate root finding, and back-substitution.

Firstly, just a little mathematical terminology: the *ideal* generated by the set of polynomials F is the set of all linear combinations of polynomials from F (combined using polynomial coefficients). A *basis* for an ideal I is a set of polynomials F that generate I. We write $I = \text{ideal}(F)$.

Thus, if $F = \{2x, x + y\}$, then the ideal I generated by F contains the polynomials $2x$, $x + y$, $10(2x) = 20x$, $(x^2+x+y)(x+y) = x^3+x^2y+x^2+2xy+y^2$, $(23x+y^{10})(2x)+(42y)(x+y) = 2(23x^2+xy^{10}+21xy+21y^2)$, and so on. Notice, that if r is a simultaneous root of $F = \{f_1, f_2, \ldots\}$ then r is a root of $g_1 f_1 + g_2 f_2 + \cdots$ for any polynomials g_i. Hence, if r is a root of the set of polynomials F, it is a root of the ideal I generated by F.

The idea is that the ideal captures the infinite number of polynomials with simultaneous roots the same as F, and somewhere inside I are the "simple" polynomials we want to find.

7.2.1 Buchberger's algorithm

The Buchberger algorithm itself is quite simple, but we need various words to help us name parts.

We need the the concept of "leading monomial" of a polynomial. For a univariate polynomial it should be clear what we mean, but for multivariate polynomials it may be a little less clear. We must introduce an order on our terms so that the term highest in the order on a polynomial will be named the leading term.

There are many ways to do this, but it is convenient (indeed, necessary!) to have an order that is compatible with multiplication. By this we mean if we have two monomials t_1 and t_2, with $t_1 < t_2$, then $tt_t < tt_2$ for any other non-zero monomial t; and also $1 < t$ for any non-constant monomial t. Examples of compatible orders: suppose $x > y > z$. Then we have lexicographic: $x^2y^3z > y^{10}$; or total degree, then lexicographic: $y^{10} > x^2y^3z > y^6$; or total degree, then inverse lexicographic: $y^{10} > y^6 > x^2y^3z$. Each of these is popular, and has specific uses.

So now we pick a fixed ordering, $<$, say (lexicographic is popular). Then the leading monomial of f is simply the largest monomial with respect to $<$.

So let us suppose we have a fixed order on monomials in mind. Now, a polynomial f is *reduced* with respect to the polynomial g if the leading monomial of g does not divide the leading monomial of f. This means that we cannot kill off the leading monomial of f by subtracting a multiple of g. More generally, f is reduced with respect to the (finite) set of polynomials F if it is reduced with respect to each $g \in F$.

We have also the related idea of *completely reduced.* We say f is completely reduced with respect to g if no monomial of f (leading or otherwise) is divisible by the leading monomial of g. Thus, we cannot reduce *any* part of f by subtracting multiples of g. Similarly for being completely reduced with respect to F.

We now have enough to state what a Gröbner basis is:
A set of polynomials G is a *Gröbner basis* for the set of polynomials F, if, $\mathrm{ideal}(F) = \mathrm{ideal}(G)$ (i.e., F and G have the same set of roots), and every reduction of an $f \in F$ to a polynomial that is reduced with respect to G is always 0. Further, G is *completely reduced* if each $g \in G$ is completely reduced with respect to $G\backslash\{g\}$.

Unwrapping this, this says the following: pick a polynomial f from our set F. Reduce this repeatedly with respect to G, and if G was a Gröbner basis for F, this will always end up at 0.

This looks like a very strange beast, but consider what is happening. Take $f \in F$. We can reduce this against G, which means we can subtract multiples of polynomials in G from f, and eventually get 0. Thus $f = h_1g_1 + h_2g_2 + \cdots + h_rg_r$, where the $g_i \in G$, and the h_i are polynomials. So now, if we have a solution $(x_0, y_0, \ldots)$ of G, this must also be a solution of f. But f was arbitrary in F, so $(x_0, y_0, \ldots)$ must be a solution of F.

What's more, since $\mathrm{ideal}(F) = \mathrm{ideal}(G)$, all the roots of F are also roots of G. This is in great contrast to the resultant method: with the Gröbner basis, we get all and exactly the roots of F.

7.3 Buchberger Algorithm

So how do we find such a G? The algorithm below was devised by Buchberger, who named it the Gröbner basis algorithm in honour of his supervisor. Many people nowadays call it the Buchberger algorithm.

Firstly another definition: let f and g be non-zero polynomials, with leading monomials f_p and g_p. Set $h = \mathrm{lcm}(f_p, g_p)$. The *S-polynomial* of f and g is

$$S(f, g) = \frac{h}{f_p}f - \frac{h}{g_p}g.$$

This is just cross-multiplying to eliminate the leading monomials.

We can now state Buchberger's algorithm:

Given a set of polynomials F, find a Gröbner basis.

1. Let $G = F$, and C be the set of all pairs of polynomials from G.
2. While C is non-empty do
   (a) Let (f_1, f_2) be a pair in C, and form the S-polynomial $h = S(f_1, f_2)$.
   (b) Reduce $h \to h'$ with respect to G.
   (c) If $h' \neq 0$, then set $C = C \cup$ new pairs from (h', G), and put $G = G \cup \{h'\}$. Set $C = C - \{(f_1, f_2)\}$.
3. Return the Gröbner basis G.

All we do is search through for pairs whose S-polynomial does not reduce to zero, and if we find one, reduce it with respect to all the others, and add it into the set.

We have a remarkable theorem:
The above is an algorithm (i.e., it terminates).

There are several proofs of this, ranging over several branches of mathematics, but the important point to note that the polynomials in the Gröbner basis G are formed by adding or subtracting linear combinations of polynomials from F. This means that any polynomial in G is in ideal(F), and in particular ideal(G) = ideal(F). Thus all the roots of G are roots of F, as was claimed.

An example: find the intersections of the circle $x^2 + y^2 = 1$ with the hyperbola $xy = 1$. We have just two polynomials

$$x^2 + y^2 - 1 \tag{1}$$
$$xy - 1 \tag{2}$$

and we shall use the lexicographic order on terms. Taking (1) against (2), we form the S-polynomial

$$y(1) - x(2) = x + y^3 - y \tag{3}$$

which we cannot reduce any further against (1) or (2). We now have the pairs (1) and (3), and (2) and (3) to consider. The S-polynomial of (1) and (3) is

$$\begin{aligned} (1) - x(3) &= (-y^3 + y)x + y^2 - 1 \\ &\to 0 \end{aligned}$$

when reduced against (2). Thus this time we have nothing to add to our basis. Now doing (2) against (3):

$$(2) - y(3) = -y^4 + y^2 - 1 \tag{4}$$

which reduces no further. We now try (2) against (4):

$$\begin{aligned} y^3(2) + x(4) &= (y^2 - 1)x - y^3 \\ &\to -y^5 + y^3 - y \end{aligned}$$

by reduction against (3)

$$\to 0$$

by reduction against (4). We really ought to check all the other pairs, but this turns out to be a complete Gröbner basis. It is not reduced, as both (1) and (2) will reduce against (3). In fact both reduce to zero, so the reduced Gröbner basis is the set $\{x + y^3 - y, y^4 - y^2 + 1\}$.

7.3.1 Optimisations

There are many short cuts and optimisations that can be added to the above basic algorithm. We mention just a few here; the first few are just heuristics, whereas the last is a theorem.

- before adding a new polynomial to G, completely reduce it with respect to the current G. This keeps down some of the expression swell.
- if adding h to G, completely reduce all other polynomials in G with respect to h. This will eliminate those polynomials that are multiples of h.
- when choosing the next pair (f_1, f_2) to compute the S-polynomial, choose the pair with the smallest LCM of leading monomials. This "means" that the leading monomials of f_1 and f_2 have a large overlap, and so, we hope, we are likely to get a "better" reduction.
- if the leading monomials of f_1 and f_2 are coprime, we may discard the pair (f_1, f_2) with no further examination: their S-polynomial will reduce to 0.

This last is based on the following theorem.

Buchberger's 2^{nd} criterion.
If gcd(leading monomial(f), leading monomial(g)) $= 1$, then $S(f, g) \to 0$.

The proof of this turns out to be quite easy. Write $f = \text{lm}(f) + f'$, where $\text{lm}(f)$ is the leading monomial of f. Similarly for g.

Since $\gcd(\text{lm}(f), \text{lm}(g)) = 1$, we have

$$S(f, g) = \text{lm}(g)f - \text{lm}(f)g = \text{lm}(g)f' - \text{lm}(f)g'.$$

Now, assuming the leading monomials of f' and g' do not cancel in the last equation, we may consider the polynomials $\text{lm}(g)f'$ and $\text{lm}(f)g'$ separately. But we may reduce $\text{lm}(g)f'$ with respect to g to find

$$\text{lm}(g)f' \to (-g')f',$$

and, likewise for the second,

$$\text{lm}(f)g' \to (-f')g'.$$

Hence,

$$S(f, g) = \text{lm}(g)f' - \text{lm}(f)g' \to (-g')f' - (-f')g' = 0.$$

It remains to show that the leading monomials of f' and g' did not cancel. So suppose, on the contrary, that $\text{lm}(g)\text{lm}(f') - \text{lm}(f)\text{lm}(g') = 0$. Then $\text{lm}(f)/\text{lm}(g) = \text{lm}(f')/\text{lm}(g')$. The fraction on the LHS is in its lowest terms (due to the criterion), thus we must have $\text{lm}(f)|\text{lm}(f')$. But this cannot happen, due to the ordering on monomials.

There is also Buchberger's 1^{st} and 3^{rd} criteria, but they are a bit more fiddly to use. (The 3^{rd} states that if there is an $h \in G$ such that $\text{lm}(h)$ divides $\text{lcm}(\text{lm}(f), \text{lm}(g))$, and we have already considered $S(f, h)$ and $S(h, g)$, then $S(f, g) \to 0$.)

7.4 Solution of Equations

If we use a lexicographic order, the final reduced Gröbner basis often will be "triangular," i.e. some equations will involve x alone, other will involve x and y alone, more will have x and y and z, and so on. In addition there is a theorem: a set of polynomial equations has finitely many zeros if, and only if, each variable appears alone (i.e. in the form $x^n + \cdots$) as one of the leading monomials of the lexicographic Gröbner basis. Thus, if we have this shape of basis, we know that there are no infinite sets of solutions to the equations. This is just the sort of information that chemists and biochemists want to about the sets of equations that describe their complex reactions.

If our equations happened to be inconsistent, the Gröbner basis would have reduced to $\{1\}$. This indicates the solution set of our polynomials is the same as the set of zeros of the set $\{1\}$, i.e., there are none.

The Buchberger is sometimes likened to the process of Gaussian elimination of linear equations. This is a little misleading, as it is a great deal more powerful, but it is true that for linear polynomials the algorithm is exactly Gaussian elimination: the S-polynomial is a linear combination that removes the leading monomial, and with a lexicographic order, we get a triangular system.

7.4.1 Notes on Gröbner Bases

You can try out the Buchberger algorithm in Reduce, using the syntax

```
groebner( { x^2 + y^2 - 1, x*y - 1 }, { x, y } );
```

The first argument is a list of polynomials, the second a list of the variables to reduce by (this allows parameters in the polynomial list, e.g., `groebner( { x^2 + y^2 - a, x*y - b }, { x, y } );` to find the intersection of two families of curves). You may also change the default term order, which is lexicographic: see the documentation for details.

Buchberger theory has many applications, solution of equations being just one example. Theorem proving can be done by encoding theorems as polynomials, and reducing the set of polynomials to see if they are consistent. Uses are being found in solid modelling: an object is modelled by a set of polynomials, and intersections of objects can be determined by the intersection of the polynomials. People have even found ways of computing GCDs and factorizing polynomials using Gröbner bases.

7.5 Wu's algorithm

For a while it seemed that Gröbner bases were the solutions to all the world's problems. However, it turns out that in practice Buchberger's algorithm can take a very long time to terminate, and the intermediate equations in the basis may be absolutely huge. There are many tricks you can apply to the basic algorithm to make it go orders of magnitude faster (the optimisations above being examples), but there are simple sets of equations that can take a (seemingly) arbitrary long time to compute. The algorithm is very sensitive to the input, too: in practice, a small change in input data can make an immense difference in the running time of a problem. For example, to solve $f = g = h = k$, we might try groebner$(f - g, g - h, h - k)$ or groebner$(f - g, f - h, f - k)$, and these may have wildly differing running times.

The problem with Gröbner bases, it seems, that they are *too* powerful objects to be considering. If all we need is a triangular set of equations in order to solve our problem, it may well be that there is an alternative formulation that could suit us better. This is exactly what Wu's algorithm gives us.

7.5.1 Definitions

Just as for Buchberger, we need to name some concepts, including an order on polynomials. Nomenclature may not correspond exactly to the previous sections.

let us suppose we have variables x_1, x_2, ..., ordered so that $x_i < x_j$ whenever $i < j$. We define the *class* of a polynomial f to be the largest subscript i of an x_i appearing in f. In this case x_i is the *leading variable* of f. If f is constant, it has class 0.

The *degree* of f is the highest degree of the leading variable in f. (This is just the degree of f in $x_{\text{class}(f)}$.) A constant has degree 0.

We can write f in the form

$$f = I x_c^d + \text{lower degree terms},$$

where $c = \text{class}(f)$, $d = \deg(f)$, and the leading coefficient I is called the *initial* of f.

A polynomial g is *reduced* with respect to f if the highest degree of x_c in g (if any) is less than d. Here we cannot subtract a multiple of f from g to reduce it.

Now we come to orders on polynomials. We can define a partial order on polynomials as follows:

$$\begin{aligned} f < g \quad \text{whenever} \quad & \text{class}(f) < \text{class}(g) \\ \text{or} \quad & \text{class}(f) = \text{class}(g) \text{ and } \deg(f) < \deg(g). \end{aligned}$$

Thus $x_1^3 + 1 < 2x_2$, and $x_2 + x_1^2 < x_2^2 + 1$. Not all polynomials are comparable, we write $f >< g$ for this case. An example: $x_2^2 + x_1 + 1 >< 2x_2^2 + x_1^2 + 2$.

An *ascending set* F of polynomials is a finite sequence of polynomials $f_1, f_2, \ldots, f_r$, such that either $\text{class}(f) = 0$ and $r = 1$, or

$$0 < \text{class}(f_1) < \text{class}(f_2) < \ldots < \text{class}(f_r),$$

and each f_j is reduced with respect to each lesser f_i.

We can extend the above order to give an order on ascending sets. Let F and G be ascending sets of lengths r and s respectively. We define $F < G$ whenever there is
either a $j \le \min\{r, s\}$ such that $f_i >< g_i$ for all $i < j$, but $f_j < g_j$,
or $r > s$ and $f_i >< g_i$ for all $i \le s$.

In words, this is: $F < G$ if either of two cases holds. The first case is when f_i and g_i are incomparable up to a point j, but $f_j < g_j$. The second case is when F is longer than G, and each g_i is incomparable to the corresponding f_i. Notice that, for this case, $F < G$ when F is *longer* than G. Again, we write $F >< G$ for incomparable ascending sets.

Examples:

$$\begin{aligned} (x_1, x_2, x_3^2) &< (x_1 - 1, x_3) \\ (x_1, x_2, x_3^2) &< (x_1, x_2) \\ (x_1, x_2, x_3^2) &>< (x_1 - 1, x_2 + 1, 2x_3^2 + x_3) \end{aligned}$$

Now, given an ascending set F, and any polynomial g we get by successive pseudo-division

$$I_1^{s_1} I_2^{s_2} \ldots I_r^{s_r} g = \sum q_i f_i + R,$$

where the I_i are the initials of the f_i, the s_i are integers, and the q_i and R are reduced with respect to all f_i. If we choose the s_i to be as small as possible (so the divisions are not *full* pseudo divisions), then R is unique, and is called the *remainder* of g with respect to F.

We need just one more definition: given a set of non-zero polynomials $P = \{p_1, p_2, \ldots p_m\}$, an ascending set of lowest order of polynomials from P is called a *basic set* for P. We can find a basic set as follows.

Given a set of polynomials P, pick a basic set.

1. Let $B = \emptyset$.
2. for $i = 1$ to the maximum class do
3. pick a polynomial of lowest degree in x_i that has x_i as leading variable, and is reduced with respect to each polynomial in B; add it into B. If none such exists, continue.

In other words, we (partially) order P with respect to $<$, and take the first reduced polynomial in each variable.

It is an easy exercise to prove this is a basic set. Basic sets are useful for the solution of the equations as they are triangular in form, by construction. Thus as we step up the basic set, we introduce more variables. The problem is, however, that the basic set can be quite short, perhaps containing just one polynomial, or does not determine the complete set of solutions. In the next section we show how to produce increasingly long basic sets.

7.5.2 Characteristic Sets

With all these definitions, the algorithms follow simply. Firstly we want to determine certain sets, called *characteristic sets,* which are triangular in form, and are part of the final solution.

Given a set of polynomials P, return an ascending set, called the *characteristic set* of P.

1. Choose a basic set B of P.
2. Form the set of remainders of polynomials in $P \backslash B$ with respect to B.
3. If there are any non-zero remainders, adjoin them to P, and return to step 1.
4. Otherwise B is the characteristic set.

(This must terminate, as the basic sets produced are decreasing, and everything in sight has the descending chain property.)

The characteristic set is triangular by construction, and every polynomial in $p \in P$ is of the form

$$Ip = \sum q_i b_i,$$

where the q_i are polynomials, I is a product of powers of initials of the b_i, and the $b_i \in B$. Hence every root of B is also a root of P, except, possibly, when that root is also a root of $\mathrm{Initials}(B)$, the set of initials of B.

On the other hand, from the construction of B, every root of P is a root of B.

So, thus far, we know the set of roots of P are pretty much the roots of B, but we still have the proviso of $I \neq 0$ to consider. How do we find those other roots of P for which the initial I *does* happen to be zero? This is simple, it is just Wu applied to $P \cup \{I\}$. We recursively call Wu to investigate this case. It may produce more roots, or it may proclaim that the set $P \cup \{I\}$ has no roots.

Hence

$$\mathrm{Zeros}(P) = \mathrm{Zeros}(B/I) \cup \bigcup \mathrm{Zeros}(P \cup \{I_i\}),$$

where I_i is the initial of b_i, and $I = \prod I_i$. We write $\mathrm{Zeros}(B/I)$ for the set of zeros of B subject to the condition $I \neq 0$. So the zeros of P are those zeros of B which are *not* zeros of I, together with the zeros of $P \cup \{I_i\}$. The last term captures those zeros that are simultaneously solutions to P and the initials I_i.

We recursively apply the characteristic set algorithm to each $P \cup \{I_i\}$. This must terminate as the basic sets get longer (i.e., decrease in the $<$ order) at each step. We stop pursuing a branch whenever we get $\mathrm{Zeros}(P \cup \{I_i\}) = \emptyset$.

This is Wu's algorithm.

Given a set of polynomials P, return a set of ascending sets of solutions.

1. Let $Z = \emptyset$, and $C = \{P\}$.
2. While C is non-empty do
   (a) Pick a set of polynomials P' from C, and put $C = C \backslash P'$.
   (b) Let B the result of applying the characteristic set algorithm to P'.
   (c) Put $Z = Z \cup \{B\}$, and $C = C \cup \bigcup \{P' \cup \{b_i\}\}$, where the b_i are those initials of B which are non-constant.
3. Return Z.

The solution set is of the form

$$\mathrm{Zeros}(P) = \bigcup \mathrm{Zeros}(B/I),$$

where the B are ascending sets, and the I are the products of the initials of the corresponding B.

7.5.3 Solution of Equations

After all this work, what have we gained? Well, we have written the zeros of P as a union of zeros of ascending sets. But the ascending are triangular, and therefore are amenable to easy solution.

Here is an example. Apply Wu to $P = \{x^2 + y^2 - 2, xy - 1\}$, where $y < x$ (viz., $y = x_1$, and $x = x_2$).

First use the characteristic set algorithm. A basic set is $B = \{xy - 1\}$, and the remainder of $x^2 + y^2 - 2$ with respect to B is $y^4 - 2y^2 + 1$. We add this to the set, and pick a new basic set, $B = \{y^4 - 2y^2 + 1, xy - 1\}$. The remainder of $x^2 + y^2 - 2$ with respect to B is now 0. Thus we return $B = \{y^4 - 2y^2 + 1, xy - 1\}$. This has initials 1 and y. So $Z = \big\{\{y^4 - 2y^2 + 1, xy - 1\}/y\big\}$.

Next we consider the set $\{x^2+y^2-2, xy-1, y\}$, where we have added in the initial y to P. Using characteristic sets again, a basic set is just $B = \{y\}$. However, the remainder of $xy - 1$ with respect to B is -1. This proves that the set $\{x^2 + y^2 - 2, xy - 1, y\}$ is inconsistent, i.e., has no solutions.

This time we return a characteristic set $\{1\}$. There are no more initials to consider, so the final result is

$$\big\{\{y^4 - 2y^2 + 1, xy - 1\}/y\big\}.$$

More generally, we get several ascending sets, each with conditions.

7.6 Buchberger vs. Wu

It is difficult to say which is the better of the two algorithms. If you require a Gröbner basis for some of its more interesting properties, then you must use Buchberger. However, if all you want is to triangularize some equations, then you might consider Wu. It is easy to find problems on which one algorithm is superior to the other; and it is just as easy to find more problems where the superiority is reversed. There is the added difficulty of comparing different implementations: Buchberger has been worked upon intensively, and many improving modifications made. On the other hand, the Wu algorithm is relatively new, and criteria analogous to, say, Buchberger's 2nd have yet to be discovered.

8 Advanced Algorithms II—Integration

8.1 Introduction

We are going to spend a little time now on the subject of integration in finite terms. This means *indefinite* integration, which is a purely algebraic process, as opposed to *definite* integration, which has a large analytic content (particularly when we have infinite limits on the integral). Integration, which is just the inverse operation to differentiation, is very much harder. Differentiation can be easily implemented in an algebra system, and if often given as an introductory undergraduate exercise in Lisp—but integration is a whole new ball game.

We say indefinite integration is an *algebraic* operation—why is this, when we were taught all about it in an analysis course? Well, consider the definition of a differential field.

A *differential field* is a field K with an operator, normally denoted by a prime, such that

- $(a \pm b)' = a' \pm b'$, and
- $(ab)' = a'b + ab'$, for all a, $b \in K$.

A typical example is $Q(x)$, such that $x' = 1$. We can deduce all the usual bits and pieces about differentiation, like $1' = (1.1)' = 1'1 + 1.1' = 1' + 1'$, so $1' = 0$. Hence $2' = (1+1)' = 0$, and in general $n' = 0$. If $b \neq 0$, then $(a/b)' = (a'b - ab')/b^2$, and so on. We call an element $c \in K$ a *constant* if $c' = 0$: the collection of all constants forms a subfield, the *constant subfield.*

So we can recast the integration problem this way: given $f \in K$, does there exist a $g \in K$ with $g' = f$? We call g an *integral*, or *anti-derivative* of f, and note that g is not unique. In fact, we like to be little more general than this, and define *integration problem* to be

given two classes of functions A and B, find an algorithm that, given an element a of A finds an element b of B such that $b' = a$, or proves that no such element exists in B.

Thus if we have $A = B = Q(x)$, and $a = 1/x^2$, the result is $b = -1/x$, but if $a = 1/x$, the answer is "impossible." If B were to be $Q(x, \log x)$, the answer is clearly $\log x$.

8.2 Naïve Methods

In school we learnt loads of ad-hoc methods of integration:

- pattern matching: this means we recognize, say, $\sin x$, and we know to replace this by $-\cos x$, or we go look up in a table that $\int 1/\sqrt{1-x^2}\,dx = \arcsin x$. However, if we come across an integral that is not in our table, we are stuck.
- substitutions: given $\int \sin x/\sqrt{1-x^2}\,dx$ we "guess" that the substitution of $x = \cos u$ might be useful. This is where a lot of people become unstuck—they are unable to do the guessing accurately. And the problem might be sufficiently complicated that you haven't got any hope of finding a usable substitution.

- integration by parts: when the above does not work, we still have integration by parts—this is just the counterpart of the product rule for differentiation. Of course, the problem now is to choose the right parts (remember $\int \log x \, dx$?), and taking care we do not go down some infinite path of sub-integrals. (Or, alternatively, recognizing we are doing this, and making use of the information so gained.)

- decomposition into partial fractions, and so on.

These are all *heuristic* methods—and algebra systems have been programmed using vast arrays of tricks and table of integrals, examples being SAINT, and SIN. Their major problem is that they did not answer the second part of the integration problem, if we fail to find an integral, is that just our fault, or is there really no such integral? The heuristics certainly do not prove the latter. We want an *algorithmic* method.

In the sequel, we shall be outlining various algorithms that solve the integration problem—or, more precisely, that solve the problem for certain classes A and B. We will cover polynomials, rational functions, logarithms and exponentials (which includes trigonometric functions), but not *algebraic* functions such as $\sqrt{x}$. There are very important reasons why $\sqrt{x}$ is harder to integrate than $\exp x$ associated with the algebraic/transcendental nature of the field extensions involved.

8.3 Polynomials

Polynomials are easy to integrate using the following rules:

- $\int x^n = x^{n+1}/(n+1)$
- $\int (a+b) = \int a + \int b$.

These solve the problem for $A = B = Q[x]$. Note that we need to prove the second rule, but this follows from the definition of differentiation: the rule really means "if $u' = a$, and $v' = b$, where u and $v \in K$, then $(u+v)' = a+b$." Note if either of u or v do not exist, this says nothing about their sum: consider $\int e^{x^2} + \int -e^{x^2}$. We shall see later that neither of the individual integrals exist, but the integral of their sum does.

We can extend this a little to $A = Q[x, x^{-1}]$, $B = Q[x, x^{-1}, \log x]$ (i.e. polynomials with possibly negative exponents) into polynomials in x and $\log x$ by

- $\int 1/x = \log x$.

We are being a bit generous in our definition of B, as we never need more than $\log x$ (we never need $\log^2 x$ or higher). In fact, this is always true.

8.4 Liouville's Theorem

All the following algorithms are based on *Liouville's theorem*, which says essentially that we never need to introduce more than log terms when integrating. We need some notation to describe the theorem accurately.

Let K be a differential field. Then θ is *elementary* over K if

1. θ is algebraic over K, i.e. is a root of some polynomial equation with coefficients in K, or
2. there exists a $\eta \in K$ such that $\theta' = \eta'\theta$; we call θ an exponential, and write $\theta = \exp \eta$, or

3. there exists a $\eta \in K$ such that $\theta' = \eta'/\eta$; we call θ an logarithm, and write $\theta = \log \eta$.

Again, logarithms and exponentials are not unique. We use $\log f$ to mean "any logarithm of f," and similarly for $\exp f$.

An *elementary extension* $L = K(\theta_1, \ldots, \theta_n)$ is one where each θ_i is elementary over $K(\theta_1, \ldots, \theta_{i-1})$.

Liouville's theorem is this:
let K be a differential field, and $\alpha \in K$ such that $y' = \alpha$ has a solution $y \in L$, where L is some elementary extension of K with the same constant subfield. Then there exist $v_1, \ldots, v_n \in K$ and constants $c_1, \ldots c_n \in K$ such that

$$\alpha = v_0' + \sum_{i=1}^{n} c_i v_i'/v_i.$$

Untangling this, Liouville's theorem says if we have an integral at all in *any* elementary extension of K (so we can add on any amount of logs and exps in any fashion), then it transpires that all we needed was logs of terms already in K, and then

$$\int \alpha = v_0 + \sum c_i \log v_i.$$

This is a rational part, plus some logarithms. Thus, if an integration can be done, the answer is really very simple.

Liouville's theorem is mainly an existence proof: we assume the existence of an integral, and show it can be re-written in the above form. So it does not help a great deal in actually finding the integral—but it does cut down drastically where we must look to find the answer. It is possible, though a little tedious, to derive a proof that there is no elementary integral of $\int e^{x^2}$ directly from the theorem.

8.5 Rational functions

Now we turn to the problem of finding an integral. We have disposed of the case where our integrand is a polynomial, or a simple sum of powers of x and x^{-1}, and the next simplest case is that of rational functions: integrate p/q, where p and q are (proper) polynomials.

Firstly we may assume that $\deg p \le \deg q$ (we always take degrees with respect to the variable of integration), for otherwise we can extract a polynomial part, and that will integrate separately (subject to the provisos noted above). Thus $(x^2+1)/(x+1) = x - (x-1)/(x+1)$. Henceforth, p/q will always be a proper rational function.

The school method of integrating p/q would now decompose this into partial fractions, and hope that each term was recognisable as an integral. However, suppose q factorizes into linear factors

$$q = \prod_{i=1}^{n} (x - a_i)^{e_i},$$

(over the complex field C, say), then we can write the partial fraction decomposition

$$\frac{p}{q} = \sum_{i=1}^{n} \frac{b_i}{(x - a_i)^{e_i}},$$

where the b_i are polynomials of degree less than e_i. By extracting the polynomial parts of the b_i, this simplifies to

$$\frac{p}{q} = \sum_{i=1}^{n} \sum_{j=1}^{e_i} \frac{b_{ij}}{(x - a_i)^j} = \sum_{i=1}^{n} \frac{b_{i1}}{x - a_i} + \sum_{i=1}^{n} \sum_{j=2}^{e_i} \frac{b_{ij}}{(x - a_i)^j}$$

for some constants b_{ij}. (For example, $x/(x-1)^2(x-2) = (-2x+1)/(x-1)^2 + 2/(x-2) = -1/(x-1)^2 - 2/(x-1) + 2/(x-2)$.)

This form we can integrate directly, using the above rules for $R[x, x^{-1}]$:

$$\int \frac{p}{q} = \sum_{i=1}^{n} b_{i1} \log(x - a_i) - \sum_{i=1}^{n} \sum_{j=2}^{e_i} \frac{b_{ij}}{(j-1)(x-a_i)^{j-1}}.$$

This also proves Liouville's theorem for the rational function case. The v_i are the args of the logs, and v_0 is the double sum.

This algorithm is fine up to a point—the point at which we have to do the partial fraction decomposition. Furthermore, when presented with an integral like $\int (5x^4+1)/(x^5+x+1)^2$, we have the problem of factorizing the denominator into linears, and this is technically a very difficult problem if we refuse to move to approximate factorizations over C. What's more the answer in this case is simply $-1/(x^5+x+1)$, so we are introducing a great deal of intermediate complexity (no pun intended!) when the answer contains no such complexity. What we need is a *rational* algorithm, that is an algorithm that only introduces those quantities that are necessary for expressing the result. We cannot always avoid such quantities, an example being $\int 1/(x^2-2) = (\log(x-\sqrt{2}) - \log(x+\sqrt{2}))/2\sqrt{2}$. We cannot write this integral without introducing $\sqrt{2}$ somewhere.

8.6 The Rational Part

Our aim is to write $\int p/q$ in the form $p_1/q_1 + \int p_2/q_2$, where the remaining integral gives only a sum of logarithms when resolved. Comparing this with the computation above, we see that q_2 is precisely the product of all the terms in the factorization of q to multiplicity one, and that q_1 is the product of the remaining terms. This suggests we should look at a square-free decomposition of q. So put $q_1 = \gcd(q, q')$, and $q_2 = q/\gcd(q, q')$. Thus

$$\begin{aligned} \frac{p}{q} &= \left(\frac{p_1}{q_1}\right)' + \frac{p_2}{q_2} \\ &= \frac{p_1'}{q_1} - \frac{p_1 q_1'}{q_1^2} + \frac{p_2}{q_2} \\ &= \frac{p_1' q_2 - p_1 s + p_2 q_1}{q} \end{aligned}$$

where $s = q_1' q_2 / q_1$ is a polynomial (the division is exact). So we must solve

$$p = p_1' q_2 - p_1 s + p_2 q_1,$$

for p_1 and p_2 in terms of the known p, s, q_1 and q_2. But the degrees of p_1 and p_2 are less than those of q_1 and q_2 respectively, and the above equation reduces to an exactly determined set of linear equations in the coefficients of the p_i. This way of splitting p/q is attributed to Horowitz, and is very suitable for computer implementation.

Here is an example. Let the proper rational part be $p/q = (x+1)/x^2(1+x^2)$. Thus $p = x+1$, and $q = x^2(1+x^2)$. We compute $q_1 = \gcd(q, q') = \gcd(x^2(1+x^2), 2x(1+2x^2)) = x$, and $q_2 = q/q_1 = x(1+x^2)$. Next, $s = q_1' q_2 / q_1 = 1.x(1+x^2)/x = 1+x^2$. So we want to solve

$$\begin{aligned} p &= p_1' q_2 - p_1 s + p_2 q_1 \quad \text{or} \\ x+1 &= p_1' x(1+x^2) - p_1(1+x^2) + p_2 x, \end{aligned}$$

or, putting $p_1 = a$ and $p_2 = b + cx + dx^2$ for some constants a, b, c, and d (as $\deg p_1 < \deg q_1$, and $\deg p_2 < \deg q_2$), this becomes

$$\begin{aligned} 1 + x &= 0 - a(1 + x^2) + (b + cx + dx^2)x \\ &= -a + bx + (-a + c)x^2 + dx^3. \end{aligned}$$

We can compare coefficients of the powers of x, and conclude $a = -1$, $b = 1$, $c = -1$, and $d = 0$. Thus $p_1 = -1$, and $p_2 = 1 - x$.

We can check the answer:

$$\begin{aligned} \left(\frac{p_1}{q_1}\right)' + \frac{p_2}{q_2} &= \left(\frac{-1}{x}\right)' + \frac{1 - x}{x(1 + x^2)} \\ &= \frac{1}{x^2} + \frac{x - x^2}{x^2(1 + x^2)} \\ &= \frac{1 + x}{x^2(1 + x^2),} \end{aligned}$$

as required.

From this we may deduce

$$\int \frac{1 + x}{x^2(1 + x^2)} = \frac{-1}{x} + \int \frac{1 - x}{x(1 + x^2)},$$

and that the latter integral is purely logarithmic.

8.7 The Logarithmic Part

So now we have p/q, whose integral consists purely of a sum of logs.

Suppose

$$\int \frac{p}{q} = \sum_{i=1}^{n} c_i \log v_i,$$

where the c_i are constants, and we may assume the v_i are polynomials, since we can replace $\log(r/s)$ by $\log r - \log s$. The v_i can be taken square-free and relatively prime, as the square-free decomposition $\log \prod r_i^i = \sum i \log r_i$ shows. Also we can take all the c_i to be distinct: $\log(x^2) + \log((x + 1)^2) = 2 \log x + 2 \log(x + 1) = 2 \log(x(x + 1))$.

Differentiating the above integral,

$$\frac{p}{q} = \sum_{i=1}^{n} \frac{c_i v_i'}{v_i}.$$

Since the v_i are square-free there can be no cancellation between v_i' and v_i, and as they are relatively prime, there can be no cancellation in the summation (the denominators are relatively prime). This implies that the v_i must be exactly the factors of q, i.e. $q = \prod_{i=1}^{n} v_i$.

Write $u_i = \prod_{j \neq i} v_j$, the product of the factors apart from v_i. Then $q' = (\prod v_i)' = \sum v_i' u_i$. Clearing the denominators in the above, we get $p = \sum c_i v_i' u_i$. Then we have

$$\begin{aligned} v_k &= \gcd(0, v_k) \\ &= \gcd(p - \sum c_i v_i' u_i, v_k) \\ &= \gcd(p - c_k v_k' u_k, v_k) \end{aligned}$$

as all the other terms are divisible by v_k, then

$$= \gcd(p - c_k \sum v_i' u_i, v_k)$$

for the same reason, and this

$$= \gcd(p - c_k q', v_k).$$

Now, on the other hand, if $l \neq k$, then

$$\begin{aligned} \gcd(p - c_k q', v_l) &= \gcd\left(\sum c_i v_i' u_i - c_k \sum v_i' u_i, v_l\right) \\ &= \gcd(c_l v_l' u_l - c_k v_l' u_l, v_l) \end{aligned}$$

as the other terms are divisible by v_l, and now this is just

$$= 1$$

as v_l is square-free, and coprime to u_l.

Putting this all together,

$$\begin{aligned} \gcd(p - c_k q', q) &= \gcd(p - c_k q', \prod v_i) \\ &= \prod \gcd(p - c_k q', v_i) \end{aligned}$$

as the v_i are relatively prime,

$$= \gcd(p - c_k q', v_k)$$

as the other terms are unity, and now this is

$$= v_k.$$

And now, conversely, suppose $c \neq$ any c_k. Following the GCD through again:

$$\begin{aligned} \gcd(p - cq', q) &= \prod_k \gcd(\sum c_i v_i' u_i, -c \sum v_i' u_i, v_k) \\ &= \prod_k \gcd(\sum (c_i - c) v_i' u_i, v_k) \\ &= \prod_k \gcd((c_k - c) v_k' u_k, v_k) \end{aligned}$$

as v_k divides all the other terms in the sum, and so this

$$= 1,$$

as we chose c distinct from all the c_k, and each v_k is coprime to $v_k' u_k$.

Hence the c_k are exactly those values of y for which $\gcd(p - yq', q) \neq 1$. And, given the c_k we can compute the corresponding v_k.

8.8 The Resultant, revisited

Recall the resultant, and Sylvester's matrix. This was originally used to solve polynomial equations, but the resultant has more interesting properties. There is a theorem:

$$\text{resultant}(f, g) = 0 \quad \text{if and only if} \quad \gcd(f, g) \neq 1.$$

In particular, the resultant of f and g is zero when they have a common factor. One way of seeing this is to consider Gaussian elimination on the Sylvester matrix: subtracting a multiple of one row from another is the same as one step in the Euclidean algorithm for the computation of $\gcd(f, g)$. We get a zero determinant if at some stage there is a linear dependence of rows, i.e. f and g have some common factor. See an appendix of DST for details.

This is exactly what we need to solve the problem of the logarithms.

8.9 The Logarithmic Part, Concluded

So now we know how to find the c_k. They are just those values of y for which $\gcd(p - yq', q)$ is non-trivial, which means the y are the roots of the equation $\text{resultant}_x(p - yq', q) = 0$, which is a polynomial equation in y.

An example: $\int 1/(x^2 - 2)\, dx$, so $p = 1$, and $q = x^2 - 2$. This turns out to have no rational part, and so we look at the resultant

$$\begin{aligned} \underset{x}{\text{resultant}}(p - yq', q) &= \underset{x}{\text{resultant}}(1 - 2xy, x^2 - 2) \\ &= \begin{vmatrix} -2y & 1 & 0 \\ 0 & -2y & 1 \\ 1 & 0 & -2 \end{vmatrix} \\ &= 1 - 8y^2 \end{aligned}$$

And so the vital values of y are $y = \pm 1/2\sqrt{2}$. And then the v_i are

$$\begin{aligned} \gcd(p - (1/2\sqrt{2})q', q) &= \gcd(1 - x/\sqrt{2}, x^2 - 2) \\ &= x - \sqrt{2} \end{aligned}$$

for $y = 1/2\sqrt{2}$ (the GCD need only be found up to a scalar multiple), and

$$\begin{aligned} \gcd(p - (-1/2\sqrt{2})q', q) &= \gcd(1 + x/\sqrt{2}, x^2 - 2) \\ &= x + \sqrt{2} \end{aligned}$$

for $y = -1/2\sqrt{2}$.

Thus

$$\int \frac{1}{x^2 - 2} = \frac{1}{2\sqrt{2}}(\log(x - \sqrt{2}) - \log(x + \sqrt{2})).$$

8.10 Integration of Logarithms and Exponentials

We can now progress to the integration of rational functions in logs and exps, in much the same way as above.

Here is an outline for the logarithmic case: we assume we have an integration algorithm for $K \to K(\log)$ (e.g. $K = Q(x)$), and θ is an elementary logarithm over K: $\theta' = \eta'/\eta$, for some $\eta \in K$. We develop an algorithm for $K(\theta) \to K(\theta, \log)$ as follows. Suppose we wish to integrate $f \in K(\theta)$. We split f into a polynomial part and a proper rational part $f = p + q/r$, where p, q, and $r \in K[\theta]$. First we prove, using Liouville's theorem, that we may integrate the polynomial part and rational parts separately to give the integral of their sum.

Next we attack the polynomial part of the integral by supposing it is of the form $\hat{p} = \sum B_i\theta^i$, differentiating, and comparing to the actual polynomial part $p = \sum A_i\theta^i$, where the A_i and B_i are in K. This offers us a differential relation between the A_i and B_i, the solution of which only requires integration in K. If at any point we find an sub-integral that does not have an elementary integral, we deduce that the whole original integral has no elementary integral.

The rational part proceeds just as before, and we can use Horowitz, noting that $'$ still means d/dx. It turns out that we do not need to consider differentiation with respect to θ, merely x again.

For the logarithmic part (i.e. the *new* logarithms), we show that we can apply the same methods we used for the rational function case, and we are led to consider the roots of the equation $\text{resultant}_\theta(q - yr', r)$. Now the roots of this equation must be *constants* for an elementary integral to exist, whereas in this case q and r are in general functions of x. Thus this is a much stronger condition than before. Thus, for $\int dx/x \log x$, we have $q = 1$, $r = x\theta$, so $\text{resultant}_\theta(1 - y(1+\theta), x\theta) = x(y-1)$. This has constant root, namely $y = 1$, so the value of the integral is $\log\theta = \log\log x$.

For exponentials, we follow much the same again, but there are complications this time. The problem is that $1/\theta$ is an exponential whenever θ is: this means the term "polynomial in θ" has to be redefined to have a wider meaning than normal, which is to include powers of θ^{-1} as well as powers of θ. Additionally, we must inductively assume we know how to solve simple linear differential equations of the form $y' + fy = g$ for y in the field K. This is the *Risch differential equation*, and it arises when we try to compute the polynomial part (we get an equation of the form $\int A_i\theta^i = B_i\theta^i$, which we can rewrite as $B_i' + i\eta' B_i = A_i$).

See DST for details.

8.11 Risch-Norman

The algorithm of the previous section (the *Risch algorithm*) is difficult. It is both difficult to understand through the curious relationships of Risch DEs and integrals in towers of field, and it is also difficult to implement on a computer. In fact we know of just two full (or nearly full) implementations of the Risch algorithm—most algebra systems have an approximation, or some hackery and heuristics. Reduce uses a technique called the *Risch-Norman* method (also known as the *Risch-Norman-ffitch* method).

This method is essentially "guess what the answer should look like, differentiate your guess, then fudge the coefficients so your guess is correct." How does this conform to our wish to find a non-heuristic algorithm, with proofs of non-existence? The answer is that our "guess" is going to be very well informed (this will rely heavily on the Liouville theorem), and we can prove that if the guess is not correct, then no elementary integral exists.

Risch-Norman also has another nice property: the original Risch algorithm relied on an inductive application to a tower of fields. This means if we wanted to integrate over $Q(x, e^x, e^{e^x})$ say, we would have to recurse down into $Q(x, e^x)$ to do an integration there, which would in turn require us to recurse down to $Q(x)$. Risch-Norman, on the other hand, treats the entire field in one fell swoop. This is a great implementational simplification.

So suppose we wish to find
$$\int \frac{f(x, \theta_1, \theta_2, \ldots, \theta_n)}{g(x, \theta_1, \theta_2, \ldots, \theta_n)},$$
where f and g are polynomials in x and the θ_i, which are elementary logs and exps. By Liouville we know if the integral exists, then
$$\frac{f(x, \theta_1, \theta_2, \ldots, \theta_n)}{g(x, \theta_1, \theta_2, \ldots, \theta_n)} = \left(\frac{p}{q}\right)' + \sum \frac{c_i g_i'}{g_i}$$
with the c_i constants and p, q and g_i polynomials in x and θ_i. Also we can take the leading coefficients of the g_i to be constant. Now if θ is a log, $\theta' = \eta'/\eta$ say, and $r(\theta)$ a polynomial in θ with constant leading

coefficient, then

$$\begin{aligned} r' &= (r_k\theta^k + r_{k-1}\theta^{k-1} + \ldots)' \\ &= r_k'\theta^k + r_{k-1}'\theta^{k-1} + \ldots \\ &\quad + r_k k\eta'\theta^{k-1} + \ldots \\ &= (r_{k-1}' + k\eta'/\eta)\theta^{k-1} + \ldots, \end{aligned}$$

and so $\deg_\theta(r') = \deg_\theta(r) - 1$. (Note that if $r_{k-1}' + k\eta'/\eta = 0$, then $r_{k-1} + k\theta = c$, for a constant c, and then $\theta = (c - r_{k-1})/k \in K$, contradicting the elementary nature of θ.)

On the other hand, if θ is an exp, $\theta' = \eta'\theta$, then

$$\begin{aligned} r' &= (r_k\theta^k + r_{k-1}\theta^{k-1} + \ldots)' \\ &= r_k'\theta^k + r_{k-1}'\theta^{k-1} + \ldots \\ &\quad + r_k k\eta'\theta^{k} + \ldots \\ &= r_k k\eta'\theta^k + \ldots, \end{aligned}$$

so this time $\deg_\theta(r') = \deg_\theta(r)$.

Comparing this back to the Liouville form above, if g factorizes as $\prod g_i^{l_i}$, then q must divide $\prod g_i^{\lambda_i}$, where $\lambda = l_i$ if g_i is purely an exponential, and $\lambda = l_i - 1$ otherwise.

So the RHS is going to be of the form

$$\frac{\mathit{something}}{\prod g_i^{\lambda_i}} + \sum c_i \log g_i.$$

Let us write

$$\int \frac{f}{\prod g_i^{l_i}} = \frac{\sum u_{i_0,i_1,\ldots,i_n} x^{i_0}\theta_1^{i_1}\ldots\theta_n^{i_n}}{\prod g_i^{\lambda_i}} + \sum c_i \log g_i,$$

where the $u \in Q$ and the c_i are to be determined. But finding them is simple, for we just differentiate the above and compare with what the answer should really be. Because the θ_i are transcendental, we can equate powers, and deduce the values of their coefficients.

An example: compute $\int(\log x/x^2)\,dx$. Put $\theta = \log x$, so the integral really is $\int \theta/x^2$. Now the denominator factorizes as $(x)^2$, x is not an exponential, so we are led to consider

$$\int \frac{\theta}{x^2} = \frac{\sum u_{ij}x^i\theta^j}{x} + c\log x.$$

Differentiating this

$$\begin{aligned} \frac{\theta}{x^2} &= \frac{\sum u_{ij}(ix^{i-1}\theta^j + jx^i\theta^{j-1}/x)x - u_{ij}x^i\theta^j}{x^2} + \frac{c}{x} \\ &= \frac{\sum u_{ij}\left((i-1)x^i\theta^j + jx^i\theta^{j-1}\right)}{x^2} + \frac{c}{x} \end{aligned}$$

Thus, multiplying up,

$$\begin{aligned} \theta - cx &= \sum u_{ij}\left((i-1)x^i\theta^j + jx^i\theta^{j-1}\right) \\ &= \sum x^i\theta^j\left((i-1)u_{ij} + (j+1)u_{i,j+1}\right) \\ &= (-u_{00} + u_{01}) + (u_{11})x + \cdots \\ &\quad + (-u_{01} + 2u_{02})\theta + (2u_{12})x\theta + \cdots \\ &\quad + \cdots \end{aligned}$$

Comparing the coefficients of $x^1\theta^0$: $-c = u_{11}$. And $x^0\theta^1$ shows $1 = -u_{01} + 2u_{02}$, so $u_{01} = -1$, as $u_{02} = 0$ (since otherwise we have an infinite regress of non-zero terms). Further, the constant term implies $0 = -u_{00} + u_{01}$, or $u_{00} = -1$. Wlog, $u_{10} = 0$, for this is just the constant of integration. Putting this together,

$$\begin{aligned} \int \frac{\log x}{x^2} &= \frac{-1 - \theta - cx\theta}{x} + c\theta \\ &= -\frac{1 + \log x}{x} \end{aligned}$$

Risch-Norman also allows us to prove the non-existence of elementary integrals, such as $\int e^{x^2}\,dx$. Write $\theta = e^{x^2}$, so $\theta' = 2x\theta$. The integral is $\int \theta$. There is no denominator, so we only have to consider the rational part

$$\int \theta = \sum u_{ij} x^i \theta^j$$

or

$$\begin{aligned} \theta &= \sum i u_{ij} x^{i-1}\theta^j + 2j u_{ij} x^{i+1}\theta^j \\ &= \sum x^i \theta^j \left((i+1)u_{i+1,j} + 2j u_{i-1,j}\right) \end{aligned}$$

Considering $x^0\theta^1$, this requires $1 = u_{11} + 2u_{-1,1}$. Well, $u_{-1,1}$ must be zero, but this leads to an infinite regress of non-zero terms. Thus the integral cannot exist. (By which we mean, of course, no *elementary* integral exists.)

The Risch-Norman method as presented above is unfortunately not an algorithm—the suspect part is where we made the degree bounds of the factors in the answer. We also need degree bounds for the numerator; in the example above we noticed that we would get an infinite regress of terms if a certain coefficient was non-zero. Unfortunately, we can not always spot this, and so we must know in advance how many terms there may exist in the numerator. There exist arbitrarily pathological cases where the number of terms can increase without limit. Consider, for example

$$\int e^{x + n\log x},$$

where n is some fixed integer. This integral has $n+1$ terms. The problem, of course, was we wrote $x^n \exp x$ in a particularly silly way. If we require that in the argument of an exponential no logarithm should appear with an integral coefficient, we are safe from this eventuality. There are a few other cases in a similar vein, and if we always ensure we write our integrals such they are *normalised* (i.e., we avoid the silly ways of writing the integrands), we can apply Risch-Norman with a great deal of confidence.

8.12 Notes on Integration

8.12.1 Structure Theorems

The above algorithms require *elementary* extensions, in particular a log or exp must be transcendental over the field below. This is not always apparent from its form: we can recognize that $e^{(\log x/2)}$ is not transcendental over $Q(x)$, but how do we spot this in general? The answer is that we can prove some *structure theorems* that do exactly this for us.

Suppose we have a field $K = Q(x, e_1, e_2, \ldots, l_1, l_2, \ldots)$, where the e_i are exponentials, and the l_j are logarithms, with $e_i' = f_i' e_i$, and $l_j' = g_j'/g_j$. If $z \in K$, when is $\log z$ independent? It turns out that if we can write

$$z = c \prod e_i^{n_i} \prod g_j^{m_j}$$

for some constant c, and rational numbers n_i and m_j, then z fails to be elementary over K.

Similarly, if we can write

$$z = c + \sum n_i f_i + \sum m_j l_j,$$

then $\exp z$ is not elementary over K.

In some sense, this is exactly what we expect, and it just so happens that these capture *all* the possible relationships between logarithms and exponentials.

8.12.2 Harder Functions

We have not touched at all the classes of algebraic functions (what is $\int \sqrt{x}$?), or the so-called "higher functions." These latter functions are themselves defined by integrals (just as exp is!), an example being the error function

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int e^{-x^2}.$$

What, now is $\int \operatorname{erf}(x^2)$? And what might we mean by an "elementary integral"? And even worse, there are the combinations of elementary and algebraic functions $\int e^{\sqrt{x}}$ and so on. This last problem has recently been addressed by Bronstein in his thesis, as has the problem of higher functions by Knowles in *his* thesis.

9 Miscellany

9.1 Richardson's Theorem

Much of what has gone before is not strictly true—or rather, there are some hidden catches in the small print that we haven't mentioned. We always implicitly assumed we were able to tell a constant from a variable, and that we know when a function was identically zero or not. In practical cases, of course, such things are fairly easy to do, but in general this may not be possible, even theoretically. This is the content of Dan Richardson's theorem: there exist well-behaved functions such that we cannot tell if they are identically zero. Not being able to distinguish these makes life extremely difficult. For example, if $f(x)$ is a function such that we cannot tell where it is zero, how can we compute $1/f(x)$?

To prove the existence of such functions, we start from a well-known theorem in number theory on the existence of solutions to Diophantine equations. A *Diophantine equation* is typically a multivariate polynomial (or some simultaneous polynomials), and the game is to discover *integral* solutions to our equation (or equations). Thus we might want integral solutions x and y to $x^2 - 2y^2 = 1$. In this case there are an infinite number of solutions, but for other examples there could be just a finite number, or none at all. It is a famous theorem that there exists no general method for the solution of Diophantine equations: and in particular, there are parameterised equations (perhaps like $y^3 - x^2 = N$) for which it is undecidable whether there exist solutions as N varies over the integers.

There are four steps on the way to our conclusion.

Let $P(x_i, N)$ be such an undecidable Diophantine equation, in some variables x_i, and a parameter N. Our first step is to note that

$$P^2 + \sum \sin^2(\pi x_i) = 0$$

is a well-behaved real analytic function of the x_i for which it is undecidable as to where the real roots are as N varies (as the sigma forces the roots to be integers, and these integers must be roots of P, as $P^2 = 0$).

The next step is to construct a function that we cannot decide if it is positive or not. Consider the proposition

$$M\left(P^2 + \sum \sin^2(\pi x_i)\right) - 1 < 0.$$

Here M is a positive function of the x_i that grows "sufficiently fast." If this proposition holds, and M is so large that the bracketed term must be close to zero, we must have the x_i close to integers, and P close to 0. But by continuity if we nudge the x_i onto the integers, the value of P cannot change by more than 1/2, i.e. the nudged values must form a root of P. Thus, from above, we cannot decide if the LHS is negative.

Now, the P is a polynomial in several variables, and in the third step we reduce this to a univariate problem. Consider the curve

$$\begin{array}{rclcl} x & = & t\cos t & = & h(t) \\ y & = & t\cos t^3 & = & k(t). \end{array}$$

This is a "space-filling curve" in the sense that we get arbitrarily close to every point in the (x, y) plane as we vary t. So now we can replace

$$\exists x, y\ f(x,y) \quad < \quad 0$$

by

$$\exists t\ f(h(t), k(t)) \quad < \quad 0$$

Repeating this procedure, we can reduce the multivariate undecidable problem above by the univariate (parameterised)

$$\exists t\ H(t, N) < 0.$$

We now have an analytic function of one variable H that is undecidable where it is negative as the parameter N varies. Now look at

$$H - |H| = \begin{cases} \equiv 0 & \text{if } H \geq 0 \text{ everywhere} \\ = 2H & \text{if } H < 0 \text{ somewhere} \end{cases}$$

Thus it is undecidable whether $F = H - |H|$ is identically zero.

So now suppose we wanted an integration algorithm. We might think the Risch or Risch-Norman methods suffice, but what do we make of

$$\int F e^{x^2}\, dx?$$

It is undecidable whether this has an elementary integral. What we need are *explicit* functions—i.e functions that we can compute explicitly with no undecidable parts. If we restrict ourselves to this subset of functions, everything is fine, and we are able to use Risch, or whatever.

Davenport has recently shown that, for the case of integration, the notion of *Z-explicit* is enough. This means we are able to tell whether an expression is an integer or not. This is stronger than being able to tell whether an expression is zero or not, but JHD shows how it is just enough to allow us to do integration. In some sense this is a converse to Richardson's theorem.

9.2 Real root finding

We have seen that using Gröbner bases we can triangularize a set of equations to make them more amenable to being solved numerically. We are now going to look briefly at the problem of finding roots of a polynomial.

For degree four, or less, polynomials there are formulae that produce the roots of an equation given its coefficients, but a famous theorem says that no such formula exists for higher degree polynomials, so we must develop other techniques for isolating roots. We talk of root *isolation* as real numbers cannot be represented accurately by machine, and so what we generally obtain is an interval that contains the root in question. The problem, then, is to find such intervals, and to prove that they contain just one root whenever possible (i.e. we must make the interval sufficiently small).

Standard root finding algorithms, such as Newton-Raphson are fine, but subject to instability. The *Wilkinson polynomial* W is

$$W(x) = (x+1)(x+2)\cdots(x+20) = x^{20} + 210x^{19} + \cdots + 20!,$$

which has 20 real roots that are seemingly well separated. If we perturb W just a little to

$$\begin{aligned} \hat{W}(x) &= W(x) + 2^{-23}x^{19} \\ &= x^{20} + (210 + 2^{-23})x^{19} + \cdots + 20! \end{aligned}$$

(this is just tweaking the coefficient of x^{19} in the last bit of 32) we find $\hat{W}$ has just 10 real roots, and the imaginary parts of the complex roots are well away from the real axis: in the region of 0.8 to 3. A small change in the coefficients has an immense change in the roots: in fact W is a very unstable polynomial.

We shall restrict ourselves to polynomials of the form $p = \sum_0^n a_i x^i$ where the a_i are integers (which we can get by simply clearing denominators). Also, we assume that p has no repeated factors, so that every root is simple.

9.2.1 Sturm Sequences

Let us define a polynomial remainder sequence as follows:

$$\begin{aligned} p_0 &= p \\ p_1 &= p' \\ p_i &= -\text{remainder}(p_{i-2}, p_{i-1}) \\ p_k &= \text{constant} \end{aligned}$$

This is very similar to the remainder sequence we use for computing GCDs, apart from the changes of sign.

Now let a be not a root of p. The *variation* $V(a)$ at a is the number of variations of sign in the sequence $p_0(a)$, $p_1(a)$, $\ldots$, $p_k(a)$. (When counting variations of sign, we ignore zeros in the sequence.) This is enough to state Sturm's theorem:

If a and b are not roots of p, and $a < b$ then the number of real roots of p in the interval (a, b) is $V(a) - V(b)$.

We define $V(\infty)$ to be the number of changes of sign in the leading coefficients of the Sturm sequence, and $V(-\infty)$ to be the same for the Sturm sequence of $p(-x)$.

Now, if we can find an upper bound on the roots of p, we have an algorithm for isolating the roots of p. Count the number of roots in the interval (a, b). If zero or one, we have a bounding interval. If more than one, we can bisect and count the roots in each half. Iterating, we can get to any desired precision.

So here is a bound on the roots of p:

$$|\alpha| \le 1 + \max\left(\left|\frac{a_{n-1}}{a_n}\right|, \left|\frac{a_{n-2}}{a_n}\right|, \ldots, \left|\frac{a_0}{a_n}\right| \right)$$

where α is a root of p. This inequality, by Gauss, is one of many (see DST).

9.2.2 Differentiation

Another method of root isolation relies on Rolle's theorem: between every pair of roots of p there is a root of p'.

Suppose we want to find the roots of p in the interval $[a, b]$. Let $\alpha_0 = a$, $\alpha_1 = \beta_1$, $\alpha_2 = \beta_2$, $\ldots$, $\alpha_r = \beta_r$, $\alpha_{r+1} = b$, where the β are the roots of p' in $[a, b]$ (which we find by recursion).

We use the following fact: if $p(\alpha_i)$ and $p(\alpha_{i+1})$ are of opposite sign, then there is exactly one root between α_i and α_{i+1}. This follows directly from Rolle, and the definition of the α.

We can now isolate roots much as before: once we have found a bound, apply the above and we get the required intervals.

9.2.3 Descartes' rule of signs

There is a further, very old root finding algorithm, based on a principle found by Descartes.

The number of sign variations of the sequence p_n, p_{n-1}, $\ldots$, p_0 (the coefficients of p), exceeds by an even non-negative integer the number of positive roots of p. This means the number of sign changes is larger than the number of roots in $(0, \infty)$, and is of the same parity.

Thus if the number of sign changes is zero, there are no positive roots. If there is one sign change, there is exactly one real positive root. If there are two or more sign changes, we split $(0, \infty) = (0, 1) \cup (1, \infty)$, and by a simple change of coordinates we transform $(0, 1) \to (0, \infty)$ and $(1, \infty) \to (0, \infty)$, and count the sign on each part.

This will isolate the positive roots of p: the negative roots are the positive roots of $p(-x)$.

9.2.4 Comparisons

Theoretically, Sturm has order like $O(n^5 \log \ldots)$, but in practice it runs in about $O(n^4)$. The differentiation method has complexity $O(n^9 \log^3 \ldots)$, but appears to run in $O(n^3)$. Lastly, Descartes method looks like $O(n^9 \ldots)$, but runs in $O(n^3)$. Theory and practice seem well divided here.

9.3 Algebraic Number Theory

Suppose we have a polynomial, $x^2 - 2$, say, and we wish to work some computations using one of its roots. Needless to say, were we to use a real approximation to the root, we would be letting ourselves in for all kinds of errors, rounding being only the first of many. For example, if we had another polynomial $x^2 - 8$, how can we tell if it factorizes over $Q(\sqrt{2})$?

The answer is to use *symbolic* roots. Thus for $\sqrt{2}$ we would introduce a new symbol, α perhaps, together with a rule $\alpha^2 \to 2$. This rewrite allows us to give a canonical form for expressions involving $\sqrt{2}$: we find

$$\begin{aligned} (x + \alpha)^2 &= x^2 + 2\alpha x + \alpha^2 \\ &\to x^2 + 2\alpha x + 2 \end{aligned}$$

Such a thing might be implemented in Reduce using a `let` rule. For roots of higher degree equations, such as $x^5 + x + 1$, for which there exists *no* formula in radicals, exactly the same holds

$$(x^2 + \beta^4)(x^2 - \beta + 1) = x^4 - \beta x^2 + x^2 + \beta^4 x^2 - \beta^5 + \beta^4$$

$$\rightarrow \quad x^4 + (\beta^4 - \beta + 1)x^2 + \beta^4 + \beta + 1,$$

where β is a new symbol subject to the rule $\beta^5 \rightarrow -\beta - 1$.

Arithmetic with such quantities is straightforward: addition and subtraction are trivial, multiplication is easy: just multiply out and use the rules to simplify, but division is a little harder.

One method of division is to use the extended Euclidean algorithm. For an algebraic number α we have the defining polynomial (or rule), $p(x)$, with $p(\alpha) = 0$. We can take p to be irreducible (every algebraic number has an unique *minimal* defining polynomial). Suppose $q(\alpha)$ is some expression in α for which we wish to compute $1/q(\alpha)$. E.g. we might want to find $1/(1+\sqrt{2})$, which means $q(x) = 1 + x$. We can assume $\deg q < \deg p$, or otherwise we can rewrite $q(\alpha)$. Then p and q must be relatively prime, $\gcd(p, q) = 1$. From the extended Euclidean algorithm we can find polynomials r and s with

$$r(x)p(x) + s(x)q(x) = 1.$$

But substituting α into this equation we get

$$\begin{aligned} r(\alpha)p(\alpha) + s(\alpha)q(\alpha) &= 0 + s(\alpha)q(\alpha) \\ &= 1, \end{aligned}$$

so $1/q(\alpha) = s(\alpha)$. This solves inversion, and thus division.

An example: find $1/(\alpha + 1)$, where $\alpha^5 + \alpha + 1 = 0$. Now $p(x) = x^5 + x + 1$, and $q(x) = x + 1$. We find

$$(-1)(x^5 + x + 1) + (x^4 - x^3 + x^2 - x + 2)(x + 1) = 1,$$

and so $1/(\alpha + 1) = \alpha^4 - \alpha^3 + \alpha^2 - \alpha + 2$.

We are able to find algorithms for GCDs and factorization of polynomials over algebraic number fields: in many cases the algorithms are a reduction of the problem to an equivalent problem over the integers, the solution of which relates back to the original question.